Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Maciej Niezabitowski

Nr albumu: 1064648

Dekompozycja drzewowa w teorii grafów

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2019

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Składam serdeczne podziękowania Panu dr. hab. Andrzejowi Kapanowskiemu, Promotorowi mojej pracy magisterskiej, za okazaną życzliwość, cenne uwagi merytoryczne, wszechstronną pomoc oraz poświęcony czas, dzięki którym niniejsza praca powstała w tym kształcie i formie.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych z dekompozycją drzewową grafów. Szczególną uwagę poświecono zagadnieniom dekompozycji w przypadku grafów cięciwowych. Zaimplementowane rozwiązania w formie dokładnych lub heurystycznych algorytmów bazują w części na istniejących rozwiązaniach takich jak algorytm wyszukiwania doskonałego uporządkowania wierzchołków PEO, algorytm wierzchołków simplicjalnych, generowania grafów cięciwowych, znajdowania największej kliki, znajdowania klik maksymalnych. Na wybranych przykładach pokazano proces dekompozycji drzewowej oraz działania zaimplementowanych rozwiązań. Rozwiązania te zostały przetestowane pod kątem poprawności, wydajności oraz spodziewanej złożoności obliczeniowej z użyciem standardowych modułów języka Python (timeit, unittest). W celu lepszego zrozumienia procesu dekompozycji dla wybranych grafów cięciwowych zostały wykonane rysunki ilustrujące działanie algorytmów. Dodatkowo w sesjach interaktywnych pokazano przykładowe działanie zaimplementowanych algorytmów wyznaczania szerokości drzewowej, dekompozycji oraz stosowanych algorytmów uzupełniających. Przedstawiono tematykę pokrycia wierzchołkowego, zbioru dominującego, zbioru niezależnego dla k-drzew. Całość pracy została podzielona na rozdziały opisujące poszczególne etapy powstawania pracy.

Słowa kluczowe: grafy, k-drzewo, graf cięciwowy, dekompozycja drzewowa, zbiór niezależny, zbiór dominujący, pokrycie wierzchołkowe

English title: Tree decomposition in graph theory

Abstract

Python implementation of selected graph algorithms connected with a tree decomposition is presented. Particular attention has been given to tree decomposition problems in the case of chordal graphs. The implemented solutions in the form of exact or heuristic algorithms are partly based on existing solutions, such as a search algorithm for perfect ordering: PEO Algorithm, algorytm of generation of chordal graphs finding biggest,maximum cliques. Selected examples show the process of tree decomposition and the operation of implemented solutions. Algorithms have been tested for correctness, performance and expected computational complexity using standard Python's modules (timeit, unittest). To better understand the decomposition process for selected chord graphs drawings have been made illustrating the operation of the algorithms. In addition, the interactive sessions show an example of the implemented algorithms for determining the tree width, decomposition and the complementary algorithms used. Presented subject for vertex cover, maximal dominating set and independent set for k-trees. The whole work was divided into chapters describing individual stages of work creation.

Keywords: graphs, k-tree, chordal graph, tree decomposition, independent set, dominating set, vertex cover

Spis treści

Spis rysunków

Listings

1. Wstęp

Tematem niniejszej pracy jest *dekompozycja drzewowa* grafów (ang. *tree decomposition*) [1]. Jest to pojęcie pochodzące ze współczesnej teorii grafów. Dekompozycja drzewowa przedstawia wierzchołki danego grafu G jako poddrzewa pewnego drzewa w taki sposób, że wierzchołki takiego grafu są sąsiadujące tylko wtedy, kiedy odpowiednie poddrzewa przecinają się. W ten sposób G tworzy podgraf grafu przecięć poddrzew. Pełny wykres przecięcia jest grafem cięciwowym. Dodatkowo w takiej reprezentacji każde poddrzewo wiąże wierzchołek wykresu ze zbiorem węzłów drzewa. Formalna definicja dekompozycji drzewowej zostanie podana w rozdziale 2.5.

Jednym z parametrów związanych ze wspomnianą dekompozycją jest *szerokość drzewowa* (ang. *treewidth*) [2]. Może ona być zdefiniowana na kilka równoważnych sposobów: jako rozmiar największego worka w dekompozycji drzewowej grafu (minus jeden), rozmiar największej kliki w uzupełnieniu cięciwowym grafu (minus jeden), maksymalna kolejność przystani opisująca strategię gry pogoń za ucieczką na grafie lub zbiór połączonych podgrafów, z których wszystkie dotykają się nawzajem.

Grafy o szerokości drzewowej co najwyżej k są również nazywane częściowymi k-drzewami. Oba te pojęcia, dekompozycja drzewowa oraz szerokość drzewowa, zostały wprowadzone przez kilku badaczy w latach 70-tych, często pod różnymi nazwami. Zauważono, że wiele problemów NP-trudnych dla ogólnych grafów jest rozwiązywalnych w czasie liniowym dla drzew. Zauważono również, że wiele problemów NP-trudnych dla ogólnych grafów jest rozwiązywalnych w czasie liniowym lub wielomianowym dla pewnych szczególnych rodzin grafów (grafy szeregowo-równoległe, grafy zewnętrznoplanarne, grafy Halina).

Odkryte algorytmy często korzystały z techniki programowania dynamicznego lub z metody dziel i zwyciężaj. W obu metodach łączy się rozwiązania dla podproblemów w jedno rozwiązanie dla większego problemu. Zauważmy, że dwa drzewa łączy się w jednym wierzchołku. Dwa grafy szeregowo-równoległe łączy się wykorzystując dwa wierzchołki. To można uogólnić na sklejanie grafów z wykorzystaniem łącznie k wierzchołków. Ogólnie można powiedzieć, że dekompozycja drzewowa jest to mapowanie grafu na pewne drzewo o grubości (szerokości drzewowej) k, przy czym zwykłe drzewo ma szerokość drzewową równą 1.

1.1. Znaczenie szerokości drzewowej

Szerokość drzewowa jest powszechnie używana jako parametr w parametryzowanej analizie złożoności algorytmów. *Algorytmy parametryzowane* zamiast wyrażać czas wykonania algorytmu tylko jako funkcję danych wejścio-

wych, biorą pod uwagę zależność od jednego lub większej liczby parametrów. Przykładowo czas wykonania algorytmu może wynosić $O(f(w)n^c)$, gdzie $f(w)$ jest pewną funkcją parametru w (często jest to funkcja wykładnicza), a c jest pewną stałą [3].

Praktyczne znaczenie szerokości drzewowej jest pośrednio potwierdzone przez istnienie serwisu *ToTo* [4]. Jest otwarta baza danych ważnych grafów, algorytmów, benchmarków, dostępna przez przeglądarkę internetową. Dla grafów z bazy danych obliczono dolne i górne ograniczenie na szerokość drzewową, czasem jest znana dokładna wartość tego parametru. Grafy przesyłane do serwisu w celu obliczenia szerokości drzewowej zostają zarchiwizowane w bazie dla innych użytkowników (grafy do 150 wierzchołków). Serwis umożliwia wyszukiwanie ciekawych grafów, wizualizację grafów i ich dekompozycji drzewowej. Możliwy jest import i eksport grafów w kilku popularnych formatach.

1.2. Cele pracy

Celem niniejszej pracy jest po pierwsze implementacja wybranych algorytmów dokładnych lub heurystycznych wyznaczających szerokość drzewową lub dekompozycję drzewową grafów. Po drugie, implementacja wybranych algorytmów rozwiązujących trudne problemy z teorii grafów przy wykorzystaniu wspomnianej dekompozycji. Do implementacji algorytmów będzie wykorzystany język Python [7], ponieważ przygotowany kod będzie rozszerzeniem pakietu graphtheory rozwijanego w Instytucie Fizyki UJ [8]. Dodatkowo czytelna składnia języka i bogata biblioteka standardowa pozwala na wykorzystanie kodu w edukacji do nauki algorytmów.

1.3. Plan pracy

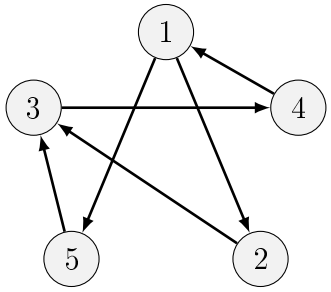
Niniejsza praca magisterska została podzielona na następujące rozdziały. Rozdział 1 jest wprowadzeniem do niniejszej pracy. Rozdział 2 definiuje podstawowe pojęcia stosowane w teorii grafów. Rozdział 3 ilustruje sposoby prezentacji grafów oraz struktury używane w bibliotece grafowej i w implementowanych algorytmach. Rozdział 4 zawiera zbiór zaimplementowanych algorytmów związanych z dekompozycją drzewową. Kod w języku Python pokazuje rozwiązania dla zagadnień takich jak: najmniejszy zbiór dominujący, najmniejsze pokrycie wierzchołkowe, największy zbiór niezależny. Rozdział 5 zawiera wnioski i podsumowanie pracy. Dodatek A prezentuje testy wydajności zaimplementowanych algorytmów dla grafów cięciwowych i ogólnych, zilustrowane wykresami czasowymi. Pokazany został wpływ współczynnika k na czas obliczeń.

2. Teoria grafów

W tym rozdziale zostaną podane podstawowe definicje i twierdzenia z teorii grafów, które będą potrzebne w opisie algorytmów grafowych.. Rysunki grafów wykonano przy użyciu pakietu *TikZ* [9].

2.1. Podstawowe definicje

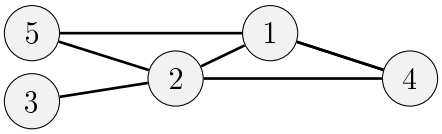
Definicja: *Graf skierowany* (ang. *directed graph*) jest to uporządkowana para zbiorów $G = (V, E)$. Pierwszy zbiór V zawiera wierzchołki należące do grafu, natomiast drugi zbiór E określa krawędzie łączące wierzchołki grafu. Krawędź łącząca dwa wierzchołki grafu jest zdefiniowana jako uporządkowana para wierzchołków (v, w), gdzie wierzchołkiem początkowym jest v, natomiast wierzchołkiem końcowym jest w. Stosowany jest również prostszy zapis vw. Powyższy zapis zilustrowany w formie graficznej najczęściej jest przedstawiany w postaci strzałki łączącej oba wierzchołki skierowanej w kierunku wierzchołka końcowego.



Rysunek 2.1. Ilustracja graficzna grafu skierowanego.

Definicja: *Graf nieskierowany* (ang. *undirected graph*) jest to para uporządkowana (V, E), składająca się ze zbioru wierzchołków V oraz zbioru E *nieuporządkowanych* par wierzchołków tworzących krawędzie. Zapis vw odpowiada jednocześnie krawędzi wv. Zatem różnica w stosunku do grafu skierowanego ogranicza się do krawędzi, które tutaj łączą dwa wierzchołki, ale nie mają kierunku.

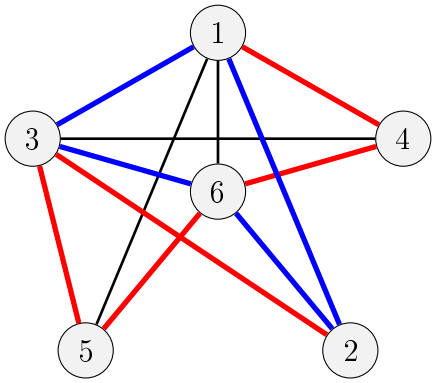
Definicja: *Ścieżką* łączącą v_0 oraz v_k nazywamy uporządkowany ciąg wierzchołków $(x_0, \ldots, x_k)$, gdzie dla każdego $i = 0, \ldots, k-1$ istnieje krawędź łącząca wierzchołek v_i z wierzchołkiem v_{i+1}. Liczbę krawędzi tworzących ścieżkę określamy jako *długość śnieżki* k. Ścieżka zawierająca niepowtarzające się



Rysunek 2.2. Ilustracja graficzna grafu nieskierowanego.

wierzchołki grafu nazywa się *ścieżką prostą*. Ścieżki są ważnym elementem teorii grafów i znalazły zastosowanie w wielu algorytmach grafowych.

Definicja: *Cykl prosty* jest to ścieżka zamknięta, w której początkowy i końcowy wierzchołek są identyczne. Najprostszym przykładem cyklu o długości jeden jest pętla. Dodatkowo każdy cykl niekoniecznie prosty może przechodzić przez powtarzające się wierzchołki grafu włącznie z wierzchołkami w pętli.



Rysunek 2.3. Rysunek przedstawia ścieżkę prostą długości $k = 5$ (kolor czerwony) oraz przykład cyklu $(3, 1, 2, 6, 3)$ (kolor niebieski).

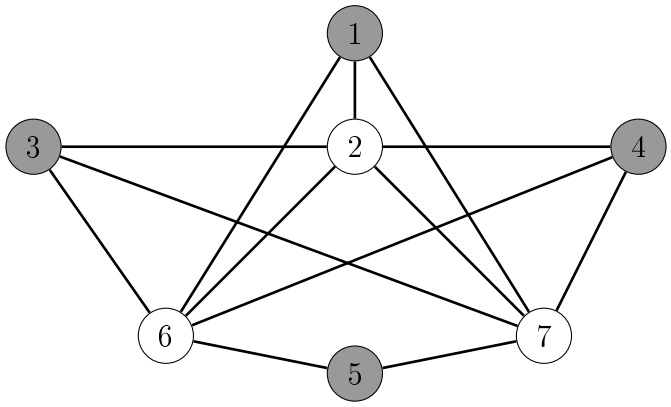
Definicja: Graf nieskierowany nazywamy *spójnym*, jeżeli pomiędzy każdą parą wierzchołków istnieje łącząca je ścieżka. Oznacza to, że startując z dowolnego wierzchołka początkowego możemy osiągnąć każdy inny wierzchołek grafu poruszając się wzdłuż istniejących krawędzi. Graf *niespójny* nie ma takiej właściwości i dzieli się na *składowe spójne*, które są podgrafami spójnymi.

2.2. Zbiory niezależne

Definicja: *Zbiorem niezależnym* (ang. *independent set*) dla grafu $G = (V, E)$ nazywamy pewien zbiór wierzchołków $S \subseteq V$ posiadający właściwość, że dla każdej pary wierzchołków v, w z tego zbioru nie istnieje krawędź w grafie G łącząca te wierzchołki. Oznacza to, że wierzchołki wchodzące w skład zbioru S nie są ze sobą parami incydentne.

Największy zbiór niezależny jest to zbiór niezależny o największej liczności. *Maksymalny zbiór niezależny* jest to taki zbiór niezależny, który nie

jest podzbiorem większego zbioru niezależnego. Największy zbiór niezależny jest jednocześnie maksymalnym zbiorem niezależnym, jednak odwrotne stwierdzenie nie w każdym przypadku musi być prawdziwe. Przykład maksymalnego zbioru niezależnego pokazano na rysunku 2.4.



Rysunek 2.4. Rysunek przedstawia graf nieskierowany dwudzielny z $n = 7$. Wierzchołki $\{1, 3, 4, 5\}$ stanowią maksymalny zbiór niezależny, a ponadto wyznaczają podział wierzchołków grafu na dwa zbiory w definicji dwudzielności. Widoczny brak incydentności pomiędzy parami wierzchołków ze zbioru niezależnego. Odpowiednie obliczenia znajdują się w rozdziale 3.2.

2.3. Pokrycia wierzchołkowe

Definicja: *Pokryciem wierzchołkowym* (ang. *vertex cover*) dla grafu $G = (V, E)$ nazywamy pewien podzbiór wierzchołków $S \subseteq V$, dla którego każda krawędź uv jest krawędzią incydentną przynajmniej z jednym wierzchołkiem należącym do zbioru S. W przypadku znalezienia pokrycia z najmniejszą liczbą wierzchołków mówimy o *najmniejszym pokryciu wierzchołkowym*.

Jest to zagadnienie często wykorzystywane w procesach wyznaczania najmniejszych pokryć, w których chcemy wykorzystać jak najmniej zasobów do alokacji, obsługi, monitoringu pewnych grup obiektów. Warto zauważyć, że dwa wierzchołki nie należące do pokrycia wierzchołkowego nie mogą mieć incydentnej krawędzi, co wynika wprost ze wspomnianej definicji, a więc taki zbiór wierzchołków tworzy zbiór niezależny. Ponadto jeśli wyznaczymy pokrycie S' będące najmniejszym pokryciem wierzchołkowym grafu G, to $V \setminus S'$ wyznacza największy zbiór niezależny grafu G oraz zbiór wierzchołków największej kliki dopełnienia grafu G. Ze wspomnianych uwarunkowań wynika, że problemy wyznaczenia najmniejszego pokrycia wierzchołkowego, znalezienia największego zbioru niezależnego, oraz wyznaczenia największej kliki dla dopełnienia grafu G są problemami równoważnymi i w ogólnym przypadku problemami NP-trudnymi. Przykład najmniejszego pokrycia wierzchołkowego pokazano na rysunku 2.5. Obliczenia znajdują się w rozdziale 3.2.

2.4. Zbiory dominujące

Definicja: *Zbiorem dominującym* (ang. *dominating set*) dla grafu $G = (V, E)$ nazywamy podzbiór D wierzchołków grafu G taki, że dla każdego wierzchołka $v \in V \setminus D$ nienależącego do zbioru D istnieje krawędź łącząca v z wierzchołkiem ze zbioru D. Podsumowując, każdy wierzchołek w grafie G należy do zbioru dominującego albo posiada sąsiada należącego do tego zbioru.

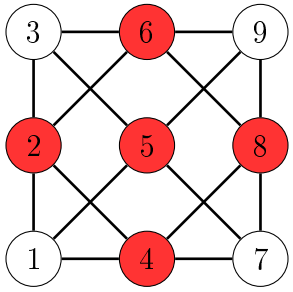
Jednym z określeń opisujących charakterystykę zbioru dominującego jest *liczba dominowania* (ang. *domination number*) $\gamma(G)$ równa liczbie wierzchołków w najmniejszym zbiorze dominującym. Problem znalezienia najmniejszego zbioru dominującego jest NP-zupełny. Przykład najmniejszego zbioru dominującego pokazano na rysunku 2.6.

2.5. Dekompozycja drzewowa grafów

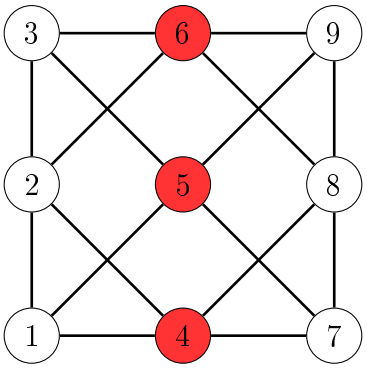
We wstępie opisaliśmy nieformalnie dekompozycję drzewową poprzez graf przecięć poddrzew pewnego drzewa. Aby zobrazować to formalnie reprezentujemy każdy węzeł drzewa (worek) jako zbiór wierzchołków powiązanych z nim. A zatem mając dany graf $G = (V, E)$ reprezentowany jako zbiór wierzchołków V i krawędzi E, dekompozycja drzewową jest to para (X, T), gdzie $X = \{X_1, \ldots, X_n\}$ jest rodziną podzbiorów zbioru wierzchołków V oraz T jest drzewem, którego węzły są workami X_i spełniającymi następujące właściwości:

1. Suma wszystkich worków X_i jest równa V, to znaczy każdy wierzchołek grafu jest związany z co najmniej jednym węzłem tego drzewa.
2. Dla każdej krawędzi grafu vw, istnieje worek X_i zawierający oba wierzchołki v oraz w. To oznacza, że wierzchołki sąsiadują ze sobą tylko wtedy kiedy odpowiednie poddrzewa mają wspólny węzeł.
3. Jeżeli dwa worki X_i oraz X_j zawierają wierzchołek v, wtedy wszystkie worki X_k drzewa na ścieżce pomiędzy X_i oraz X_j zawierają także ten wierzchołek (w drzewie istnieje dokładnie jedna ścieżka między X_i oraz X_j). To znaczy, worki związane z wierzchołkiem v tworzą spójne poddrzewo w T. Ta zależność znana jest również pod pojęciem koherencji.

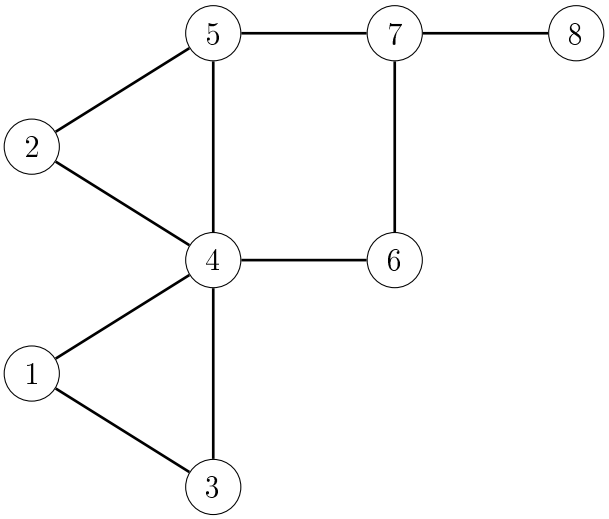
Dekompozycja ładna: Ładna dekompozycja drzewowa (ang. *nice tree decomposition*) jest rozumiana jako drzewo z korzeniem. Korzeń i liście drzewa



Rysunek 2.5. Minimalne pokrycie wierzchołkowe dla przykładowego grafu z $n = 9$.



Rysunek 2.6. Najmniejszy zbiór dominujący - liczba dominowania $\gamma(G) = 3$.

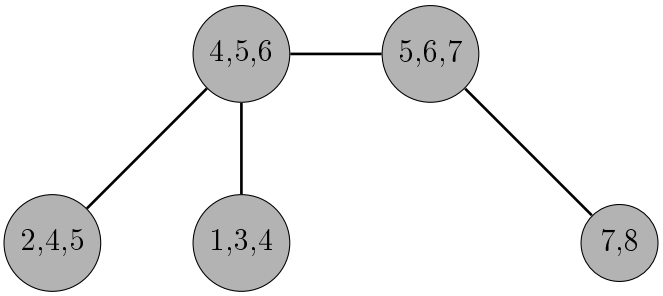


Rysunek 2.7. Przykładowy graf prosty z $n = 8$.

są pustymi workami. Każdy worek nie będący liściem jest jednym z trzech typów: worek wprowadzający (ang. *introduce node*), worek zapominający (ang. *forget node*), i worek łączący (ang. *join node*) [3]. Liczność sąsiednich worków może różnić się co najwyżej o jeden. Szczegółów nie będziemy opisywać, ponieważ nie będziemy korzystać z tej postaci dekompozycji. Wykorzystuje się ją przy dowodzeniu poprawności algorytmów działających na drzewie dekompozycji.

Stwierdzenie: Jeżeli graf G ma dekompozycję drzewową o szerokości w, to można wyznaczyć ładną dekompozycję drzewową z liczbą worków ograniczoną przez $O(wn)$.

Dekompozycja bez powtórzeń: Przez dekompozycję drzewową bez powtórzeń rozumiemy taką dekompozycję drzewową (X, T), gdzie dla każdej krawędzi ij nie zachodzi $X_i \subset X_j$. Dekompozycję drzewową z powtórzeniami można przekształcić do dekompozycji bez powtórzeń przez sklejenie worków X_i i X_j.



Rysunek 2.8. Przykładowa dekompozycja drzewowa grafu z rysunku 2.7. Obliczenia dekompozycji znajdują się w rozdziale 3.2

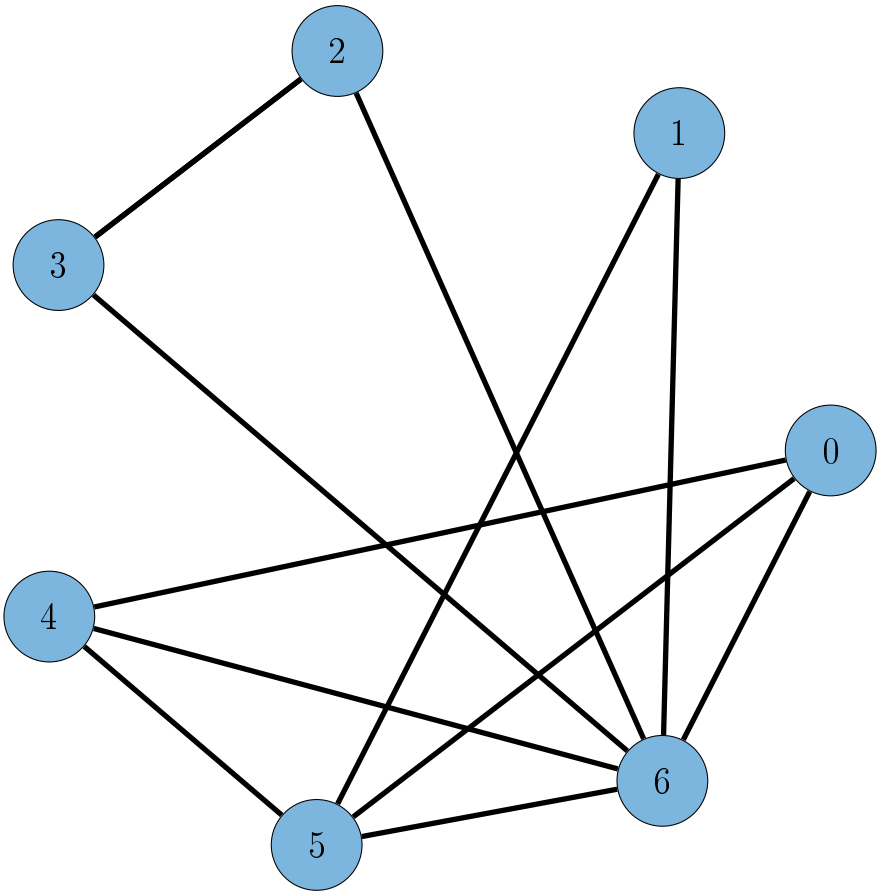
Stwierdzenie: Każda dekompozycja drzewowa bez powtórzeń grafu o n wierzchołkach zawiera nie więcej niż n worków. Dowód robi się przez indukcję po liczbie wierzchołków n.

Stwierdzenie: Jeżeli graf G ma dekompozycję drzewową o szerokości w, to liczba krawędzi grafu G jest rzędu $O(wn)$ [3]. Można to pokazać przez usuwanie z uzupełnienia cięciowowego grafu G kolejnych wierzchołków simplicjalnych, które są połączone z sąsiadami najwyżej w krawędziami.

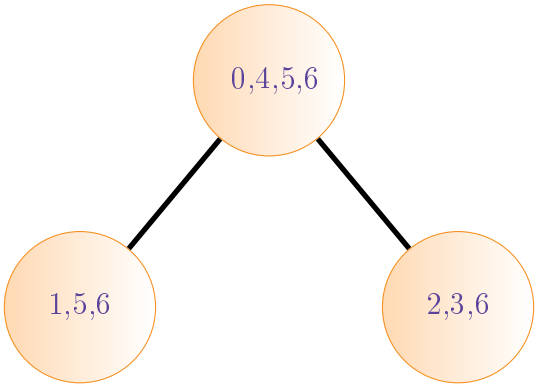
2.6. Dekompozycja drzewowa dla k-drzew

Definicja: W teoria grafów k-drzewo jest nieskierowanym grafem sformowanym rozpoczynając od $(k+1)$-wierzchołkowego grafu pełnego, a następnie powtarzalnie dodając wierzchołki w taki sposób, że każdy dodany wierzchołek v ma dokładnie k sąsiadów tworzących klikę [6]. k-drzewa są dokładniej maksymalnymi grafami z określoną szerokością drzewową, grafami do których nie może być dodanych więcej krawędzi bez zwiększenia ich szerokości drzewowej. k-drzewa zaliczamy do grafów cięciwowych, dla których maksymalne kliki są tego samego rozmiaru $k+1$ i wszystkie minimalne separatory klik są także takiego samego rozmiaru k. Jeżeli z danego k-drzewa usuniemy wybrane krawędzie tak, że szerokość drzewowa podgrafu się nie zmieni, to taki podgraf nazywamy *częściowym k-drzewem.*

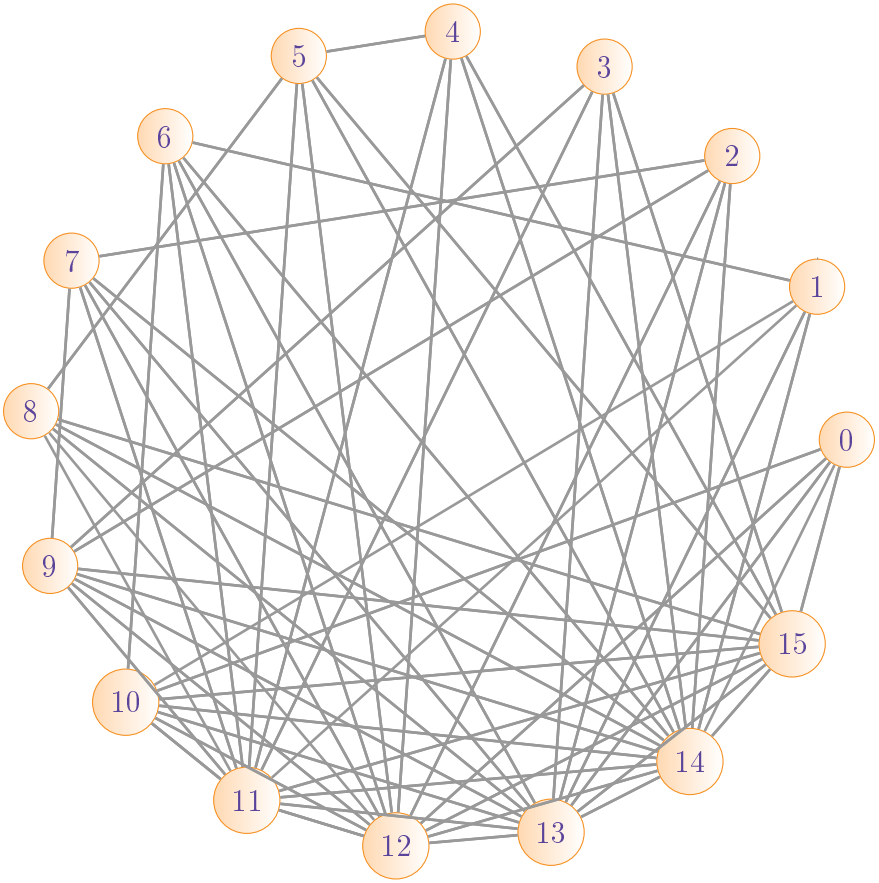
Dla danego k-drzewa G mamy $|V(G)| = n$, $|E(G)| = nk - k(k+1)/2$. Dekompozycja drzewowa T tego k-drzewa będzie miała $|V(T)| = n - k$ worków i $|E(T)| = n - k - 1$ krawędzi. Szerokość drzewowa k-drzewa i częściowego k-drzewa wynosi k.



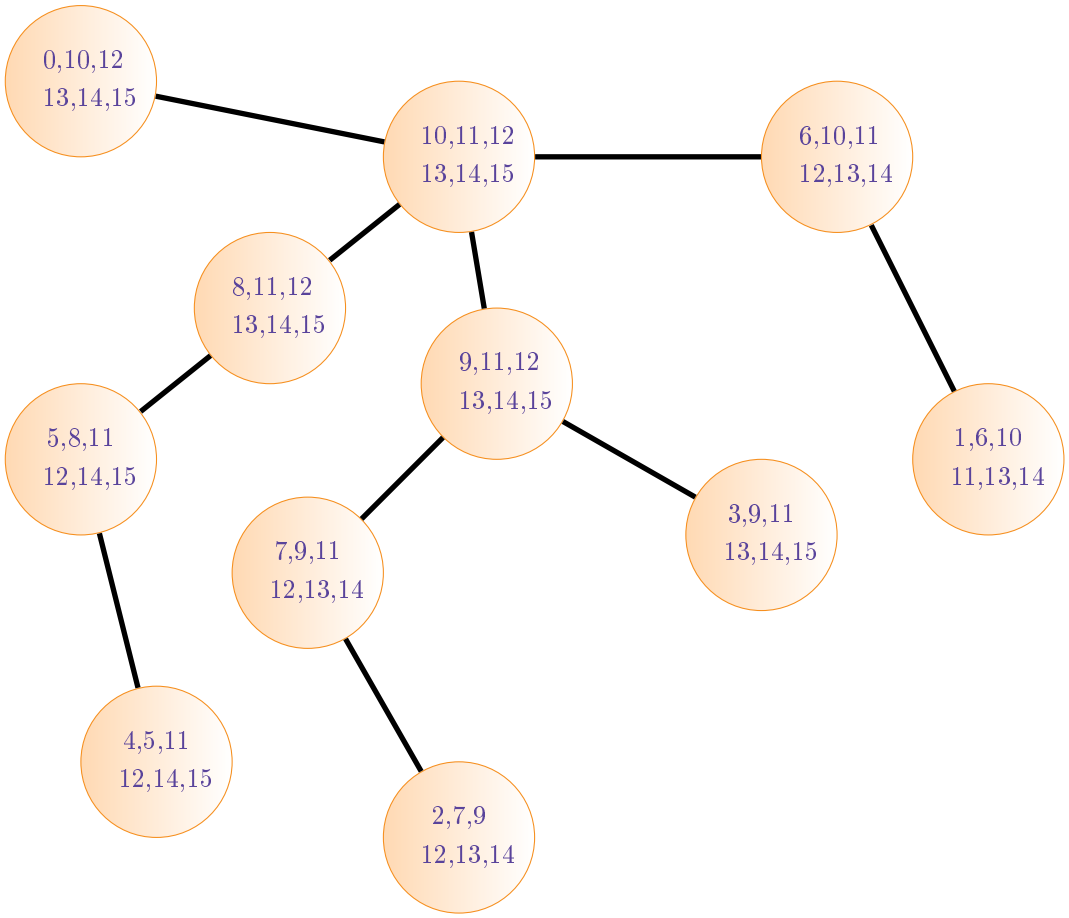
Rysunek 2.9. Graf cięciwowy, największy cykl równy 4 posiadający cięciwy, czyli krawędzie nie wchodzące w skład cyklu. Widoczne również trywialne grafy cięciwowe indukowane poprzez podgrafy.



Rysunek 2.10. Dekompozycja drzewowa grafu cięciwowego.Obliczenia znajdują się w rozdziale 3.2



Rysunek 2.11. Przykład k-drzewa, gdzie $k = 5$, $n = 16$.



Rysunek 2.12. Dekompozycja drzewowa dla k-drzewa, $k = 5$, $n = 16$, liczba worków $|V(T)| = n - k = 11$, liczba krawędzi $|E(T)| = n - k - 1 = 10$. Obliczanie dekompozycji dla k-drzewa pokazano w rozdziale 3.2.

3. Implementacja grafów

W tym rozdziale przedstawimy interfejs grafów i przykładowe obliczenia, które można wykonać za pomocą stworzonego kodu.

3.1. Struktury danych

Wierzchołek: Wierzchołkami grafu mogą być obiekty pythonowe hashowalne i porównywalne. Zwykle są to liczby lub stringi, ale w pewnych problemach są to krawędzie grafu (graf krawędziowy) lub krotki wierzchołków (dekompozycja drzewowa). Świadczy to o dużej elastyczności przyjętego interfejsu biblioteki.

Krawędź: Krawiędzie skierowane grafu są instancjami klasy Edge z modułu edges. Krawędzie są hashowalne i można je porównywać, przy czym dla krawędzi edge najpierw porównywana jest waga krawędzi edge.weight, następnie atrybuty edge.source i edge.target.

Graf: Grafy skierowane i nieskierowane, ważone i bez wag, są instancjami klasy Graph. W bibliotece dostępnych jest kilka implementacji tej klasy, ale najbardziej elastyczna implementacja zawarta jest w module graphs.

Algorytm: Proste algorytmy grafowe są implementowane jako zwykłe funkcje, natomiast dłuższe algorytmy zwykle mają postać klas o dość ujednoliconym interfejsie. Konstruktor klasy wykonuje czynności przygotowawcze. Metoda run() uruchamia właściwy algorytm. Wyniki działania algorytmu i dane początkowe są dostępne jako atrybuty w instancji klasy algorytmu.

Dekompozycja drzewowa: Dekompozycja drzewowa jest to drzewo, które przechowujemy jako instancję klasy Graph. Wierzchołkami drzewa są worki (kliki), które reprezentujemy jako posortowane krotki wierzchołków.

Doskonałe uporządkowanie wierzchołków (PEO): Jest to lista wierzchołków grafu.

Klika: Jest to zbiór wierzchołków grafu.

Zbiór niezależny: Jest to zbiór wierzchołków grafu.

Zbiór dominujący: Jest to zbiór wierzchołków grafu.

Pokrycie wierzchołkowe: Jest to zbiór wierzchołków grafu.

3.2. Przykładowe obliczenia

Algorytmy zaimplementowane w pracy można łatwo wykorzystać w praktycznych obliczeniach grafowych. Przedstawimy przykładowe sesje interaktywne z takimi obliczeniami.

Przykład 1: Obliczenia z grafami cięciwowymi.

```
>>> from edges import Edge
>>> from graphs import Graph
# Import generatorow grafow cieciwowych.
>>> from chordaltools import make_random_ktree
>>> from chordaltools import make_random_chordal
# Przykladowy graf cieciwowy.
>>> G = make_random_chordal(10)
# Wyznaczanie PEO.
>>> from chordaltools import find_peo_mcs
>>> order = find_peo_mcs(G)
# Wyznaczanie najwiekszego zbioru niezaleznego.
>>> from chordaliset import find_maximum_independent_set
>>> iset = find_maximum_independent_set(G, order)
# Budowanie drzewa dekompozycji.
>>> from chordaltools import find_td_chordal
>>> T = find_td_chordal(G, order)
# Obliczanie treewidth - I sposob (klika najwieksza).
>>> from clique1 import find_maximum_clique_peo
>>> max_clique = find_maximum_clique_peo(G, order)
>>> treewidth = len(max_clique)-1
# Obliczanie treewidth - II sposob (drzewo dekompozycji).
>>> treewidth = max(len(bag) for bag in T.iternodes()) -1
# Wyznaczanie najmniejszego zbioru dominujacego.
>>> from chordaldset import ChordalDominatingSet
>>> algorithm = ChordalDominatingSet(G, T)
>>> algorithm.run()
>>> print algorithm.dominating_set
# Wyznaczanie najmniejszego pokrycia wierzcholkowego.
>>> from chordalcover import ChordalNodeCover
>>> algorithm = ChordalNodeCover(G, T)
>>> algorithm.run()
>>> print algorithm.node_cover
```

Przykład 2: Obliczenia z dowolnymi grafami.

```
>>> from edges import Edge
>>> from graphs import Graph
# Przykladowy graf spojny.
>>> G = Graph(10)
# Budowanie grafu ...
# Wyznaczanie uporzadkowania wierzcholkow MCS.
>>> from chordaltools import find_peo_mcs
>>> order = find_peo_mcs(G)
# Budowanie drzewa dekompozycji na bazie MCS.
>>> from chordaltools import find_td_order
>>> T = find_td_order(G, order)
```

```
# Obliczanie treewidth.
>>> treewidth = max(len(bag) for bag in T.iternodes()) -1
# Budowanie drzewa dekompozycji na bazie heurystyki
# najmniejszego stopnia.
>>> from chordaltools import find_td_min_deg
>>> T = find_td_min_deg(G)
# Wyznaczanie najwiekszego zbioru niezaleznego.
>>> from tdiset import TDIndependentSet
>>> algorithm = TDIndependentSet(G, T)
>>> algorithm.run()
>>> print algorithm.independent_set
>>> print algorithm.cardinality
# Wyznaczanie najmniejszego zbioru dominujacego.
>>> from tddset import TDDominatingSet
>>> algorithm = TDDominatingSet(G, T)
>>> algorithm.run()
>>> print algorithm.dominating_set
>>> print algorithm.cardinality
# Wyznaczanie najmniejszego pokrycia wierzcholkowego.
>>> from tdcover import TDNodeCover
>>> algorithm = TDNodeCover(G, T)
>>> algorithm.run()
>>> print algorithm.node_cover
>>> print algorithm.cardinality
```

Przykład 3: Obliczenia dla grafu dwudzielnego z rysunku 2.4.

```
>>> from edges import Edge
>>> from graphs import Graph
>>> G = Graph(7)
>>> for node in range(1, 8):
...     G.add_node(node)
>>> G.add_edge(Edge(1, 2))
>>> G.add_edge(Edge(1, 6))
>>> G.add_edge(Edge(1, 7))
>>> G.add_edge(Edge(2, 3))
>>> G.add_edge(Edge(2, 4))
>>> G.add_edge(Edge(2, 6))
>>> G.add_edge(Edge(2, 7))
>>> G.add_edge(Edge(3, 6))
>>> G.add_edge(Edge(3, 7))
>>> G.add_edge(Edge(4, 6))
>>> G.add_edge(Edge(4, 7))
>>> G.add_edge(Edge(5, 6))
>>> G.add_edge(Edge(5, 7))
>>> G.show()
# Zbudowany graf z rysunku 2.4.
1 : 2 6 7
2 : 1 3 4 6 7
3 : 2 6 7
4 : 2 6 7
5 : 6 7
6 : 1 2 3 4 5
7 : 1 2 3 4 5
# Najwiekszy zbior niezalezny Golumbic PEO.
>>> from chordaliset import find_maximum_independent_set
>>> print find_maximum_independent_set(G, range(1, 8))
```

```
set([1, 3, 4, 5])
# Wyznaczanie najwiekszego zbioru niezaleznego z uzyciem dekompozycji
# na bazie heurystyki najmniejszego stopnia.
>>> from chordaltools import find_td_min_deg
>>> T = find_td_min_deg(G)
>>> from tdiset import TDIndependentSet
>>> algorithm = TDIndependentSet(G, T)
>>> algorithm.run()
>>> print algorithm.independent_set
set([1, 3, 4, 5])
```

Przykład 4: Obliczenia dla grafu z rysunku 2.5.

```
>>> from edges import Edge
>>> from graphs import Graph
# Budowa grafu nieskierowanego z n=9.
>>> G = Graph(9)
>>> for node in range(1, 10):
...     G.add_node(node)
>>> G.add_edge(Edge(1, 2))
>>> G.add_edge(Edge(1, 4))
>>> G.add_edge(Edge(1, 5))
>>> G.add_edge(Edge(2, 3))
>>> G.add_edge(Edge(2, 4))
>>> G.add_edge(Edge(2, 6))
>>> G.add_edge(Edge(3, 5))
>>> G.add_edge(Edge(3, 6))
>>> G.add_edge(Edge(4, 7))
>>> G.add_edge(Edge(4, 8))
>>> G.add_edge(Edge(5, 7))
>>> G.add_edge(Edge(5, 9))
>>> G.add_edge(Edge(6, 8))
>>> G.add_edge(Edge(6, 9))
>>> G.add_edge(Edge(7, 8))
>>> G.add_edge(Edge(8, 9))
>>> G.show()
1 : 2 4 5
2 : 1 3 4 6
3 : 2 5 6
4 : 8 1 2 7
5 : 1 3 9 7
6 : 8 9 2 3
7 : 8 4 5
8 : 9 4 6 7
9 : 8 5 6
# Znajdowanie przyblizonego drzewa dekompozycji.
>>> from chordaltools import find_td_min_deg
>>> T = find_td_min_deg(G)
# Wyznaczanie minimalnego pokrycia wierzcholkowego.
>>> from tdcover import TDNodeCover
>>> algorithm = TDNodeCover(G, T)
>>> algorithm.run()
>>> print algorithm.node_cover
set([8, 2, 4, 5, 6])
```

Przykład 5: Obliczanie dekompozycji dla grafu z rysunku 2.7.

```
>>> from edges import Edge
>>> from graphs import Graph
# Budowanie grafu prostego z n=8.
>>> G = Graph(8)
>>> for node in range(1, 9):
...     G.add_node(node)
>>> G.add_edge(Edge(1, 3))
>>> G.add_edge(Edge(1, 4))
>>> G.add_edge(Edge(2, 4))
>>> G.add_edge(Edge(2, 5))
>>> G.add_edge(Edge(3, 4))
>>> G.add_edge(Edge(4, 5))
>>> G.add_edge(Edge(4, 6))
>>> G.add_edge(Edge(5, 7))
>>> G.add_edge(Edge(6, 7))
>>> G.add_edge(Edge(7, 8))
>>> G.show()
# Wyznaczanie uporzadkowania wierzcholkow MCS.
>>> from chordaltools import find_peo_mcs
>>> order = find_peo_mcs(G)
# Budowanie drzewa dekompozycji na bazie MCS.
>>> from chordaltools import find_td_order
>>> T = find_td_order(G, order)
>>> T.show()
# Drzewo dekompozycji jako graf wazony.
# Wierzcholki to krotki, wagi sa ujemne.
(7, 8) : (5, 6, 7)(-1)
(5, 6, 7) : (7, 8)(-1) (4, 5, 6)(-2)
(4, 5, 6) : (5, 6, 7)(-2) (2, 4, 5)(-2) (1, 3, 4)(-1)
(2, 4, 5) : (4, 5, 6)(-2)
(1, 3, 4) : (4, 5, 6)(-1)
```

Przykład 6: Obliczanie dekompozycji dla grafu cięciwowego z rysunku 2.10.

```
>>> from edges import Edge
>>> from graphs import Graph
# Budowanie grafu cieciwowego z n=7.
>>> n = 7
>>> G = Graph(n)
>>> for node in range(n):
...     G.add_node(node)
>>> G.add_edge(Edge(0, 4))
>>> G.add_edge(Edge(0, 5))
>>> G.add_edge(Edge(0, 6))
>>> G.add_edge(Edge(1, 5))
>>> G.add_edge(Edge(1, 6))
>>> G.add_edge(Edge(2, 3))
>>> G.add_edge(Edge(2, 6))
>>> G.add_edge(Edge(3, 6))
>>> G.add_edge(Edge(4, 5))
>>> G.add_edge(Edge(4, 6))
>>> G.add_edge(Edge(5, 6))
>>> G.show()
0 : 4 5 6
1 : 5 6
2 : 3 6
```

```
3 : 2 6
4 : 0 5 6
5 : 0 1 4 6
6 : 0 1 2 3 4 5
# Wyznaczanie PEO.
>>> from chordaltools import find_peo_mcs
>>> order = find_peo_mcs(G)
>>> print order
[3, 2, 1, 6, 5, 4, 0]
# Dekompozycja drzewowa na bazie PEO.
>>> from chordaltools import find_td_chordal
>>> T = find_td_chordal(G, order)
>>> T.show()
# Drzewo dekompozycji jako graf wazony.
(2, 3, 6) : (0, 4, 5, 6)(-1)
(0, 4, 5, 6) : (2, 3, 6)(-1) (1, 5, 6)(-2)
(1, 5, 6) : (0, 4, 5, 6)(-2)
```

Przykład 7: Obliczanie dekompozycji dla k-drzewa z rysunku 2.12.

```
>>> from graphs import Graph
>>> from chordaltools import make_random_ktree
>>> from chordaltools import find_td_chordal
>>> n = 16
>>> G = make_random_ktree(n, 5) # generator k-drzew
>>> G.show()
0 : 10 12 13 14 15
1 : 14 10 11 13 6
2 : 9 12 13 14 7
3 : 9 11 13 14 15
4 : 11 12 5 14 15
5 : 4 8 11 12 14 15
6 : 1 10 11 12 13 14
7 : 2 9 11 12 13 14
8 : 5 11 12 13 14 15
9 : 2 3 7 11 12 13 14 15
10 : 0 1 6 11 12 13 14 15
11 : 1 3 4 5 6 7 8 9 10 12 13 14 15
12 : 0 2 4 5 6 7 8 9 10 11 13 14 15
13 : 0 1 2 3 6 7 8 9 10 11 12 14 15
14 : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 15
15 : 0 3 4 5 8 9 10 11 12 13 14
>>> order = range(n)    # PEO dla grafu z generatora
>>> T = find_td_chordal(G, order)    # tree decomposition
>>> T.show()
# Drzewo dekompozycji jako graf wazony.
(6, 10, 11, 12, 13, 14) : (1, 6, 10, 11, 13, 14)(-5)
    (10, 11, 12, 13, 14, 15)(-5)

(0, 10, 12, 13, 14, 15) : (10, 11, 12, 13, 14, 15)(-5)

(2, 7, 9, 12, 13, 14) : (7, 9, 11, 12, 13, 14)(-5)

(10, 11, 12, 13, 14, 15) : (0, 10, 12, 13, 14, 15)(-5)
    (8, 11, 12, 13, 14, 15)(-5) (6, 10, 11, 12, 13, 14)(-5)
    (9, 11, 12, 13, 14, 15)(-5)
```

(3, 9, 11, 13, 14, 15) : (9, 11, 12, 13, 14, 15)(−5)

(4, 5, 11, 12, 14, 15) : (5, 8, 11, 12, 14, 15)(−5)

(7, 9, 11, 12, 13, 14) : (2, 7, 9, 12, 13, 14)(−5)
(9, 11, 12,13, 14, 15)(−5)

(8, 11, 12, 13, 14, 15) : (5, 8, 11, 12, 14, 15)(−5)
(10, 11, 12, 13, 14, 15)(−5)

(5, 8, 11, 12, 14, 15) : (4, 5, 11, 12, 14, 15)(−5)
(8, 11, 12,13, 14, 15)(−5)

(1, 6, 10, 11, 13, 14) : (6, 10, 11, 12, 13, 14)(−5)

(9, 11, 12, 13, 14, 15) : (7, 9, 11, 12, 13, 14)(−5)
(3, 9, 11, 13, 14, 15)(−5) (10, 11, 12, 13, 14, 15)(−5)

---

4. Algorytmy

W tym rozdziale zostaną opisane algorytmy zaimplementowane w ramach niniejszej pracy. Algorytmy dotyczą budowy drzewa dekompozycji oraz rozwiązywania trzech problemów grafowych dla grafów cięciwowych i grafów dowolnych. Rozważane problemy to znajdowanie największego zbioru niezależnego, najmniejszego zbioru dominującego i najmniejszego pokrycia wierzchołkowego.

4.1. Dekompozycja drzewowa dla grafów cięciwowych

Dla grafu cięciwowego G można wyznaczyć wszystkie kliki maksymalne za pomocą algorytmu z książki Golumbica [10], którego implementacja w pracy [11] znalazła się w funkcji find_all_maximal_cliques(). Kliki maksymalne używamy do stworzenia ważonego grafu przecięć klik W_G, opisanego w pracy [12]. Optymalne grzewo dekompozycji T grafu cięciwowego G jest drzewem rozpinającym o największej wadze dla grafu W_G. Do wyznaczenia takiego drzewa możemy użyć algorytmu znajdującego minimalne drzewo rozpinające, jeżeli w grafie W_G zmienimy wszystkie wagi na ujemne. W naszej implementacji korzystamy z algorytmu Prima, a całość algorytmu jest zawarta w funkcji find_td_chordal(). Przyjmujemy, że na wejściu dany jest graf cięciwowy i jego PEO.

Wyznaczanie wszystkich klik maksymalnych grafu cięciwowego zajmuje czas liniowy $O(V + E)$. Liczba klik maksymalnych jest ograniczona przez $O(V)$. Czas budowy grafu przecięć klik szacujemy na $O(V^2 w)$, ponieważ sprawdzane są wszystkie kombinacje par worków $O(V^2)$, a w pętli sprawdzane jest przecięcie worków w czasie $O(w)$. Algorytm Prima jest ograniczony przez $O(V^2)$ lub $O(E \log V)$ zależnie od użytej implementacji. Łączny czas algorytmu szacujemy na $O(V^2 w)$.

Listing 4.1. Dekompozycja drzewowa grafu cięciwowego.

```
def find_td_chordal(graph, order):
    """Finding a tree decomposition for chordal graphs."""
    cliques = find_all_maximal_cliques(graph, order)
    H = Graph()    # graf przeciec klik maksymalnych
    bag_dict = dict()
    # Budowanie workow.
    for c in cliques:
        bag = tuple(sorted(c))
        bag_dict[bag] = c
        H.add_node(bag)
    # Budowanie krawedzi grafu przeciec klik.
    for (bag1, bag2) in itertools.combinations(bag_dict, 2):
```

```
            inter = bag_dict[bag1].intersection(bag_dict[bag2])
            if inter:
                H.add_edge(Edge(bag1, bag2, -len(inter)))
        algorithm = PrimMST(H)
        algorithm.run()
        algorithm.to_tree()
        return algorithm.mst
```

4.2. Dekompozycja drzewowa na bazie danego uporządkowania wierzchołków grafu

Dekompozycję drzewową dowolnego spójnego grafu można wyznaczyć na bazie danego doskonałego uporządkowania wierzchołków w dopełnieniu cięciwowym grafu. Niestety zwykle nie znamy takiego uporządkowania, więc używamy jakiegoś innego uporządkowania, np. uporządkowania MCS (ang. *maximum cardinality search*). Na ogół otrzymamy wtedy nieoptymalną dekompozycję drzewową i górne oszacowanie na szerokość drzewową.

Algorytm przetwarza wierzchołki zgodnie z podanym uporządkowaniem. Tworzona jest kopia grafu, ponieważ nastąpi redukcja grafu. Dla danego wierzchołka v najpierw tworzymy klikę z jego sąsiadów z $N(v)$. Dodawane krawędzie są zapisywane. Następnie wierzchołek v jest usuwany razem z krawędziami incydentnymi. Powyższe czynności są powtarzane dla wszystkich wierzchołków.

W drugiej części algorytmu zapisane krawędzie są dodawane do oryginalnego grafu, aby utworzyć dopełnienie cięciwowe. Wyznaczane jest drzewo dekompozycji tego grafu cięciwowego. Na koniec przywracana jest pierwotna postać grafu przez usunięcie dodanych tymczasowo krawędzi.

Złożoność obliczeniowa pierwszej części algorytmu wynosi $O(Vw^2)$ ze względu na sprawdzanie klik w każdym przebiegu pętli. W drugiej części dominuje czas pracy funkcji find_td_chordal(), który wynosi $O(V^2w)$. Łączny czas pracy algorytmu wynosi więc $O(V^2w)$.

Listing 4.2. Dekompozycja drzewowa na bazie danego uporządkowania wierzchołków grafu.

```
def find_td_order(graph, order):
    """Finding a tree decomposition using a given order."""
    if graph.is_directed():
        raise ValueError("the graph is directed")
    # Nie chcemy modyfikowac oryginalnego grafu.
    graph_copy = graph.copy()
    edge_list = []
    for source in order:
        # Robimy klike z {source} + N(source).
        # Dla grafow cieciwowych jest to niepotrzebne.
        for (node, target) in itertools.combinations(
            graph_copy.iteradjacent(source), 2):
            edge = Edge(node, target)
            if not graph_copy.has_edge(edge):
                graph_copy.add_edge(edge)
```

```
                edge_list.append(edge)
        # Usuwanie source z krawedziami.
        graph_copy.del_node(source)
    # Robimy graf cieciwowy z oryginalu.
    for edge in edge_list:
        graph.add_edge(edge)
    # Graf stal sie cieciwowy.
    T = find_td_chordal(graph, order)
    # Przywracanie oryginalu.
    for edge in edge_list:
        graph.del_edge(edge)
    return T
```

4.3. Dekompozycja drzewowa na bazie heurystyki najmniejszego stopnia

Heurystyka najmniejszego stopnia pozwala znaleźć górne oszacowanie szerokości drzewowej i odpowiednią dekompozycję drzewową. Najpierw ustalamy uporządkowanie wierzchołków. Z grafu usuwamy kolejno wierzchołki o najmniejszym stopniu, przy czym dodajemy potrzebne krawędzie tak, aby sąsiedzi usuwanego wierzchołka tworzyli klikę. W ten sposób otrzymujemy pewne uzupełnienie cięciwowe oryginalnego grafu wraz z PEO, dla którego wyznaczamy drzewo dekompozycji poprzednio opisaną metodą. Złożoność obliczeniową szacujemy na $O(V^3)$.

Listing 4.3. Dekompozycja drzewowa na bazie heurystyki najmniejszego stopnia.

```
def find_td_min_deg(graph):
    """Finding a tree decomposition using the minimum degree heuristic."""
    # Graf powinien byc spojny.
    if graph.is_directed():
        raise ValueError("the graph is directed")
    order = list()    # zapisuje kolejnosc usuwania wierzcholkow
    used = set()    # do szybkiego sprawdzania usunietych wierzcholkow
    # Nie chce modyfikowac oryginalnego grafu, wiec mam kopie.
    graph_copy = graph.copy()    # O(V+E) time and memory
    edge_list = []
    # W kazdym kroku usuwamy jeden wierzcholek.
    for _ in xrange(graph.v()):
        # Wybieram node o najmniejszym stopniu, nie zaliczony do order.
        source = min((node for node in graph_copy.iternodes()
            if node not in used), key=graph_copy.degree)
        order.append(source)
        used.add(source)
        # Robie klike z {source} + N(source).
        for (node, target) in itertools.combinations(
            graph_copy.iteradjacent(source), 2):
            edge = Edge(node, target)
            if not graph_copy.has_edge(edge):
                graph_copy.add_edge(edge)
                edge_list.append(edge)
```

```
        # Usuwanie source z krawedziami.
        graph_copy.del_node(source)
    # Robie graf cieciwowy z oryginalu.
    for edge in edge_list:
        graph.add_edge(edge)
    # Graf stal sie cieciwowy.
    T = find_td_chordal(graph, order)
    # Przywracanie oryginalu.
    for edge in edge_list:
        graph.del_edge(edge)
    return T
```

4.4. Największy zbiór niezależny dla grafów cięciwowych

Problem znajdowania największego zbioru niezależnego dla grafu cięciwowego ma rozwiązanie podane przez Gavrila [13]. Algorytm działa w czasie liniowym $O(V + E)$ i wykorzystuje PEO. Implementacja algorytmu w języku Python została podana w pracy Olak [11] i jest to funkcja o nazwie find_maximum_independent_set().

Algorytm przetwarza wierzchołki zgodnie z kolejnością wyznaczoną przez PEO, co odpowiada odrywaniu liści (worków) z drzewa dekompozycji. Dany wierzchołek zaliczamy do zbioru niezależnego, o ile nie jest sąsiadem wierzchołka zaliczonego wcześniej do zbioru niezależnego. Poprawność wyznaczonego zbioru niezależnego można sprawdzić funkcją is_independent_set().

Listing 4.4. Testowanie zbioru niezależnego.

```
def is_independent_set(graph, iset):
    """Testing independent sets in O(E) time."""
    for edge in graph.iteredges():
        if edge.source in iset and edge.target in iset:
            return False
    return True
```

4.5. Problem największego zbioru niezależnego

Rozwiązywanie problemu największego zbioru niezależnego dla grafów o danej dekompozycji drzewowej (X, T) i małej szerokości drzewowej w. Opiszemy intuicję która prowadzi do końcowego algorytmu. Niech S będzie optymalnym zbiorem niezależnym dla grafu G. Zbiór S będzie miał część wspólną z różnymi workami X_i, ale nie wiemy jak wygląda ta część wspólna. Każdy worek może mieć liczność co najwyżej $w + 1$, co daje 2^{w+1} możliwości (podzbiorów) do sprawdzenia. Problemy możemy rozwiązywać w różnych poddrzewach niezależnie, ale przy łączeniu rozwiązań częściowych musimy mieć zgodność rozwiązań na przecięciu worków sąsiadujących.

W końcowym algorytmie wybieramy dowolnie worek z drzewa T, który będzie korzeniem. Worki przetwarzamy w kolejności postorder (najpierw

dzieci, na końcu rodzic), a ten porządek zapewnia nam DFS. Przyjmijmy następujące oznaczenia [1].

— S to poszukiwany największy zbiór niezależny.
— D_i jest to suma worka X_i i wszystkich jego potomków.
— Niech $\{U_i\}$ oznaczają różne zbiory niezależne, zawierające się w worku X_i, $U_i \subset X_i$.
— Jeżeli X_i jest liściem, wtedy $A_i(U_i) = |U_i|$ to liczność zbioru niezależnego w zbiorze $D_i = X_i$.
— Jeżeli X_i nie jest liściem, wtedy $A_i(U_i) = |U_i| + \sum_j B_{ij}(U_i)$ to liczność zbioru niezależnego w zbiorze D_i, przy czym $B_{ij}(U_i)$ to przyczynki od dzieci.
— $B_{ij}(U_i) = \max\{A_j(U_j) - |U_i \cap U_j| : U_j \cap X_i = U_i \cap X_j, U_j \text{ niezależne w } X_j\}$, czyli jesteśmy w zbiorze D_j i chcemy dobrać takie U_j, aby jego wkład był **największy**. Ważne, że możemy dobierać tylko takie U_j, które są zgodne z U_i.
— Jeżeli X_i jest korzeniem, to wybieramy **największe** $A_i(U_i)$.

Funkcje A_i i B_{ij} pokazują wyznaczanie jedynie liczności największego zbioru niezależnego, ale łącząc zbiory U_i możemy otrzymać także cały zbiór S.

Dane wejściowe: Dowolny graf spójny G i jego dekompozycja drzewowa bez powtórzeń T.

Problem: Wyznaczenie największego zbioru niezależnego.

Złożoność: Sprawdzenie warunku $U_j \cap X_i = U_i \cap X_j$ zajmuje czas $O(w)$. W worku X_i może być $O(2^{w+1})$ zbiorów niezależnych. Liczba generowanych zbiorów niezależnych w każdym węźle wynosi $O(2^{w+1})$. Liczba worków jest ograniczona przez $O(n)$. Łączna złożoność obliczeniowa wynosi więc $O(4^{w+1}wn)$. Zauważmy, że złożoność jest liniowa w liczbie wierzchołków n, ale wykładnicza w szerokości drzewowej w. Stąd stosowanie algorytmu w praktyce jest ograniczone do grafów o małej szerokości drzewowej.

Uwagi: W implementacji zbiory U_i są tworzone za pomocą generatora wszystkich podzbiorów zbioru X_i [iter_power_set()], a następnie odrzucane są zbiory, które nie są niezależne. Niezależność zbiorów jest sprawdzana przez test istnienia krawędzi łączących dowolne dwa elementy zbioru U_i. Krawędzi do sprawdzenia będzie $O((w+1)^2)$, trzeba sprawdzić 2^{w+1} zbiorów U_i w każdym worku, więc przygotowanie poprawnych zbiorów niezależnych na starcie zajmuje czas $O(2^{w+1}(w+1)^2 n)$.

Listing 4.5. Moduł tdiset.

```
#!/usr/bin/python

import itertools
from powersets import iter_power_set
from edges import Edge

class TDIndependentSet:
```

```
    """Find a maximum independent set using a tree decomposition."""

    def __init__(self, graph, tree_decomposition):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.td = tree_decomposition
        self.parent = dict()    # for tree decomposition
        self.independent_set = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # A single connected component, a single tree.
            self.parent[source] = None    # before _visit
            arg1 = self._visit(source)
            self.independent_set.update(max(arg1, key=len))
            self.cardinality = len(self.independent_set)
        else:
            # A forest is possible. NOT FOR TD
            for bag in self.td.iternodes():
                if bag not in self.parent:
                    self.parent[bag] = None    # before _visit
                    arg1 = self._visit(bag)
                    self.independent_set.update(max(arg1, key=len))
            self.cardinality = len(self.independent_set)

    def _compose(self, top, arg1, bag, arg2):
        """Compose results."""
        result = []
        separator = set(top) & set(bag)
        for set1 in arg1:
            # Do kazdego set1 chcemy dolaczyc jak najwiecej od dziecka.
            # Mozemy dolaczyc tylko takie rozwiazania, ktore sa zgodne
            # na przecieciu workow.
            set3 = set1
            for set2 in arg2:
                #if set2 & set(top) == set1 & set(bag):
                if set2 & separator == set1 & separator:
                    # Jezeli jest zgodnosc przy przecieciu to sprawdzamy.
                    set3 = max(set3, set1|set2, key=len)
            result.append(set3)
        return result

    def _is_iset(self, subbag):
        """Test if a subbag is an iset."""
        for (source, target) in  itertools.combinations(subbag, 2):
            if self.graph.has_edge(Edge(source, target)):
                return False
        return True

    def _visit(self, top):    # top is tuple
```

```
        """Explore recursively the connected component."""
        # Funkcja zwraca liste zbiorow niezaleznych dla grafu obejmujacego
        # worek top i jego potomkow.
        # Tworze liste mozliwych rozwiazan dla bag.
        # Trzeba sprawdzic wszystkie mozliwe podzbiory.
        # Zaczynamy od zbiorow niezaleznych dla samego worka top.
        arg1 = [set(subbag) for subbag in iter_power_set(top)
            if self._is_iset(subbag)]
        # Do zbiorow niezaleznych dolaczamy wierzcholki od workow dzieci.
        for bag in self.td.iteradjacent(top):
            if bag not in self.parent:
                self.parent[bag] = top    # before _visit
                arg2 = self._visit(bag)
                arg1 = self._compose(top, arg1, bag, arg2)
        return arg1
```

4.6. Najmniejsze pokrycie wierzchołkowe dla grafów cięciwowych

Rozwiązywanie problemu najmniejszego pokrycia wierzchołkowego dla grafów cięciwowych. Stworzenie dekompozycji drzewowej dla grafu cięciwowego opisaliśmy wcześniej. Mając drzewo dekompozycji postępujemy tak jak w przypadku ogólnego grafu, więc szczegóły podamy w następnym rozdziale.

Implementacja algorytmu zawarta jest w module chordalcover. W porównaniu do ogólnego przypadku uproszczeniu ulega sprawdzenie, czy dany podzbiór worka jest pokryciem wierzchołkowym w worku. Dla grafów cięciwowych poprawnym pokryciem wierzchołkowym worka będzie cały worek lub worek z usuniętym jednym wierzchołkiem. Poprawność wyznaczonego pokrycia wierzchołkowego można sprawdzić funkcją is_node_cover().

Listing 4.6. Testowanie pokrycia wierzchołkowego.

```
def is_node_cover(graph, cover):
    """Testing node covers in O(E) time."""
    for edge in graph.iteredges():
        if edge.source not in cover and edge.target not in cover:
            return False
    return True
```

4.7. Problem najmniejszego pokrycia wierzchołkowego

Rozwiązywanie problemu najmniejszego pokrycia wierzchołkowego dla grafów o danej dekompozycji drzewowej (X, T) i małej szerokości drzewowej w. Podobnie jak w przypadku zbioru niezależnego załóżmy, że C jest poszukiwanym najmniejszym pokryciem wierzchołkowym. Zbiór C będzie miał część wspólną z różnymi workami X_i, ale jej nie znamy. Każdy worek może mieć liczność co najwyżej $w+1$, co daje 2^{w+1} możliwości (podzbiorów) do sprawdzenia. Ponadto części z różnych sąsiadujących worków X_i oraz X_j

mają wspólną część w przecięciu worków $X_i \cap X_j$. Ten warunek musi być spełniony przy łączeniu częściowych rozwiazań z różnych poddrzew.

Przyjmijmy następujące oznaczenia [1].

— C to poszukiwane najmniejsze pokrycie wierzchołkowe.
— D_i jest to suma worka X_i i wszystkich jego potomków.
— Niech $\{U_i\}$ oznaczają różne pokrycia wierzchołkowe, zawierające się w worku X_i, $U_i \subset X_i$.
— Jeżeli X_i jest liściem, wtedy $A_i(U_i) = |U_i|$ to liczność pokrycia wierzchołkowego w zbiorze $D_i = X_i$.
— Jeżeli X_i nie jest liściem, wtedy $A_i(U_i) = |U_i| + \sum_j B_{ij}(U_i)$ to liczność pokrycia wierzchołkowego w zbiorze D_i, przy czym $B_{ij}(U_i)$ to przyczynki od dzieci.
— $B_{ij}(U_i) = \min\{A_j(U_j) - |U_i \cap U_j| : U_j \cap X_i = U_i \cap X_j, U_j \text{ pokrycie w } X_j\}$, czyli jesteśmy w zbiorze D_j i chcemy dobrać takie U_j, aby jego wkład był **najmniejszy**. Ważne, że możemy dobierać tylko takie U_j, które są zgodne z U_i.
— Jeżeli X_i jest korzeniem, to wybieramy **najmniejsze** $A_i(U_i)$.

Funkcje A_i i B_{ij} pokazują wyznaczanie jedynie liczności najmniejszego pokrycia wierzchołkowego, ale łącząc zbiory U_i możemy otrzymać także cały zbiór C. Podobnie jak dla zbiorów niezależnych złożoność obliczeniowa algorytmu wynosi $O(4^{w+1}wn)$.

Listing 4.7. Moduł tdcover.

```
#!/usr/bin/python

import itertools
from powersets import iter_power_set
from edges import Edge

class TDNodeCover:
    """Find a minimum node cover using a tree decomposition."""

    def __init__(self, graph, tree_decomposition):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.td = tree_decomposition
        self.parent = dict()    # for tree decomposition
        self.node_cover = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # A single connected component, a single tree.
            self.parent[source] = None    # before _visit
            arg1 = self._visit(source)
            self.node_cover.update(min(arg1, key=len))
```

```
            self.cardinality = len(self.node_cover)
        else:
            # A forest is possible. NOT FOR TD
            for bag in self.td.iternodes():
                if bag not in self.parent:
                    self.parent[bag] = None   # before _visit
                    arg1 = self._visit(bag)
                    self.node_cover.update(min(arg1, key=len))
            self.cardinality = len(self.node_cover)

    def _compose(self, top, arg1, bag, arg2):
        """Compose results."""
        result = []
        separator = set(top) & set(bag)
        for set1 in arg1:
            # Do kazdego set1 chcemy dolaczyc jak najmniej od dziecka.
            # Mozemy dolaczyc tylko takie rozwiazania, ktore sa zgodne
            # na przecieciu workow.
            set3 = None   # set1 poszerzony o set2
            for set2 in arg2:
                #if set2 & set(top) == set1 & set(bag):
                if set2 & separator == set1 & separator:
                    # Jezeli jest zgodnosc przy przecieciu to sprawdzamy.
                    if set3 is None:
                        set3 = set1|set2
                    else:
                        set3 = min(set3, set1|set2, key=len)
            result.append(set3)
        return result

    def _is_node_cover(self, bag, subbag):
        """Test if a subbag is a node cover in the bag."""
        # Dla kazdej krawedzi z bag musimy sprawdzic, czy choc jeden
        # koniec nalezy do pokrycia.
        for (source, target) in  itertools.combinations(bag, 2):
            if self.graph.has_edge(Edge(source, target)):
                if source not in subbag and target not in subbag:
                    return False
        return True

    def _visit(self, top):
        """Explore recursively the connected component."""
        # Start from a single node.
        # Tworze liste mozliwych rozwiazan dla bag.
        # Trzeba sprawdzic wszystkie mozliwe podzbiory.
        # Zaczynamy od pokryc dla samego worka top.
        arg1 = [set(subbag) for subbag in iter_power_set(top)
            if self._is_node_cover(top, subbag)]
        # Do pokryc dolaczamy wierzcholki od workow dzieci.
        for bag in self.td.iteradjacent(top):
            if bag not in self.parent:
                self.parent[bag] = top   # before _visit
                arg2 = self._visit(bag)
                arg1 = self._compose(top, arg1, bag, arg2)
        return arg1
```

4.8. Najmniejszy zbiór dominujący dla grafów cięciwowych

Rozwiązywanie problemu najmniejszego zbioru dominującego dla grafów cięciwowych. Stworzenie dekompozycji drzewowej dla grafu cięciwowego opisaliśmy wcześniej. Mając drzewo dekompozycji postępujemy tak jak w przypadku ogólnego grafu, więc szczegóły podamy w następnym rozdziale.

Implementacja algorytmu zawarta jest w module chordaldset. W porównaniu do ogólnego przypadku uproszczeniu ulega sprawdzenie, czy dany podzbiór worka jest zbiorem dominującym w worku. Dla grafów cięciwowych każdy niepusty podzbiór worka jest zbiorem dominującym, bo worek jest kliką. Poprawność wyznaczonego zbioru dominującego można sprawdzić funkcją is_dominating_set().

Listing 4.8. Testowanie zbioru dominującego.

```
def is_dominating_set(graph, dset):
    """Testing dominating sets in O(V+E) time."""
    for source in graph.iternodes():
        if source in dset:
            continue
        covered = False
        for target in graph.iteradjacent(source):
            if target in dset:
                covered = True
                break
        if not covered:
            return False
    return True
```

4.9. Problem najmniejszego zbioru dominującego

Rozwiązywanie problemu najmniejszego zbioru dominującego dla grafów o danej dekompozycji drzewowej (X, T) i małej szerokości drzewowej w. Podobnie jak w przypadku zbioru niezależnego załóżmy, że S jest poszukiwanym najmniejszym zbiorem dominującym. Zbiór S będzie miał część wspólną z różnymi workami X_i, ale jej nie znamy. Algorytm będzie pewnym uogólnieniem rozwiązań podanych dla drzew [14] i dla grafów szeregowo-równoległych [15]. W obliczeniach należy uwzględnić częściowe rozwiązania formalnie niepoprawne, w których nie wszystkie wierzchołki są zdominowane. Musimy tak postąpić, bo pewne wierzchołki będą zdominowane dopiero przez wierzchołki pojawiające się w rodzicu danego worka lub wyżej.

Wygodnie jest rozważyć kolorowanie każdego worka trzema kolorami.

— *Wierzchołek biały* **nie** należy do częściowego rozwiązania i **nie** jest zdominowany.
— *Wierzchołek szary* **nie** należy do częściowego rozwiązania, ale jest zdominowany.
— *Wierzchołek czarny* należy do częściowego rozwiązania.

Każdy worek może mieć liczność co najwyżej $w+1$, a wtedy istnieje 3^{w+1} różnych 3-kolorowań i to jest przestrzeń stanów. Jest oczywiste, że w końcowym rozwiązaniu nie mogą wystąpić wierzchołki białe.

Części z różnych sąsiadujących worków X_i oraz X_j mają wspólną część w przecięciu worków $X_i \cap X_j$. Warunek równości czarnych wierzchołków musi być spełniony przy łączeniu częściowych rozwiązań z różnych poddrzew. Zauważmy, że wierzchołek biały w jednym worku może być wierzchołkiem szarym w sąsiednim worku, dlatego nie musi być zgodności w odniesieniu do białych i szarych wierzchołków.

Załóżmy, że do worka X_i przyłączamy poddrzewo z korzeniem w worku X_j. Niech w pewnym częściowym rozwiązaniu wierzchołek v jest biały w worku X_j, a przy tym wierzchołek v już nie występuje w worku X_i. Jest jasne, że taki wierzchołek v nie będzie mógł być zdominowany podczas uzupełniania częściowych rozwiązań i należy odrzucić częściowe rozwiązanie z wierzchołkiem v. Jest to dodatkowy warunek do sprawdzenia przy przyłączeniu częściowych rozwiązań.

Przyjmijmy następujące oznaczenia [1].

— W każdym worku X_i przygotowujemy różne startowe 3-kolorowanie wierzchołków, czyli dla wierzchołków czarnych B_i znajdujemy wierzchołki szare G_i, a pozostałe wierzchołki z worka będą białe W_i. Dla każdego 3-kolorowania zachodzi warunek $|W_i| + |G_i| + |B_i| = |X_i|$. Oznaczamy przez $\bar{W}_i$, $\bar{G}_i$, $\bar{B}_i$ zbiory wierzchołków uaktualnione wierzchołkami pochodzącymi od dzieci. W przypadku liści $W_i = \bar{W}_i$, $G_i = \bar{G}_i$, $B_i = \bar{B}_i$.
— Podczas dołączania dziecka X_j do worka X_i będziemy szukać najlepszego rozwiązania częściowego dla każdego 3-kolorowania z worka X_i. Musi być spełniony warunek $\bar{B}_j \cap X_i = \bar{B}_i \cap X_j$.
— Zbiór wierzchołków znikających przy przejściu z worka X_j do X_i wyliczamy jako $X_j \backslash X_i$, a warunek konieczny do spełnienia przy łączeniu ma postać $\bar{W}_j \backslash X_i| = 0$.
— Uaktualnione rozwiązanie częściowe (3-kolorowanie) w worku X_i będzie miało postać $\bar{B}_i = B_i \cup \bar{B}_j$, $\bar{G}_i = G_i \cup \bar{G}_j$, $\bar{W}_i = (W_i \backslash \bar{G}_j) \cup (\bar{W}_j \backslash G_i)$.
— Dla ustalonego startowego 3-kolorowania (W_i, G_i, B_i) przebiegamy wszystkie zgodne 3-kolorowania $(\bar{W}_j, \bar{G}_j, \bar{B}_j)$ i wybieramy uaktualnione rozwiązanie $(\bar{W}_i, \bar{G}_i, \bar{B}_i)$ z **najmniejszym** zbiorem $\bar{B}_i$. Tak przetwarzamy wszystkie dzieci worka X_i i robimy dalsze uaktualnienia.
— Jeżeli X_i jest korzeniem i przyłączyliśmy wszystkie jego dzieci X_j, to końcowym optymalnym rozwiązaniem jest 3-kolorowanie z **najmniejszym** $\bar{B}_i$ i $|\bar{W}_i| = 0$.

Złożoność obliczeniową algorytmu szacujemy na $O(9^{w+1}wn)$. Inicjalizacja początkowych kolorowań w każdym worku zajmuje czas $O(2^{w+1}(w+1)^2 n)$, ponieważ można wybrać 2^{w+1} zestawów wierzchołków czarnych, a znalezienie wierzchołków szarych i białych zależy od liczby krawędzi w worku i jest $O((w+1)^2)$.

Listing 4.9. Moduł tddset.

```
#!/usr/bin/python
```

```
from powersets import iter_power_set


class TDDominatingSet:
    """Find a minimum dominating set usin a tree decomposition."""

    def __init__(self, graph, tree_decomposition):
        """The algorithm initialization."""
        if graph.is_directed():
            raise ValueError("the graph is directed")
        self.graph = graph
        self.td = tree_decomposition
        self.parent = dict()   # for tree decomposition
        self.dominating_set = set()
        self.cardinality = 0
        import sys
        recursionlimit = sys.getrecursionlimit()
        sys.setrecursionlimit(max(self.graph.v() * 2, recursionlimit))

    def run(self, source=None):
        """Executable pseudocode."""
        if source is not None:
            # A single connected component, a single tree.
            self.parent[source] = None   # before _visit
            arg1 = self._visit(source)
            dset = None
            for (white, grey, black) in arg1:
                if len(white) == 0:   # tylko prawidlowe rozwiazania
                    if dset is None:
                        dset = black
                    else:
                        dset = min(dset, black, key=len)
            self.dominating_set.update(dset)
            self.cardinality = len(self.dominating_set)
        else:
            # A forest is possible. NOT FOR TD
            for bag in self.td.iternodes():
                if bag not in self.parent:
                    self.parent[bag] = None   # before _visit
                    arg1 = self._visit(bag)
                    dset = None
                    for (white, grey, black) in arg1:
                        if len(white) == 0:   # tylko prawidlowe rozwiazania
                            if dset is None:
                                dset = black
                            else:
                                dset = min(dset, black, key=len)
                    self.dominating_set.update(dset)
            self.cardinality = len(self.dominating_set)

    def _compose(self, top, arg1, bag, arg2):
        """Compose results."""
        #print "compose top bag", top, bag
        result = []
        separator = set(top) & set(bag)
        introduce = set(top) - separator
        forget = set(bag) - separator
```

```
        for (white1, grey1, black1) in arg1:
            # Do kazdego set1 chcemy dolaczyc jak najmniej od dziecka.
            # Mozemy dolaczyc tylko takie rozwiazania, ktore sa zgodne
            # na przecieciu workow.
            white3 = None
            grey3 = None
            black3 = None
            for (white2, grey2, black2) in arg2:
                if (black2 & separator == black1 & separator and
                    len(white2 & forget) == 0):
                    # Jezeli jest zgodnosc przy przecieciu to sprawdzamy.
                    # Nie moga zostac zapomniane wierzcholki niezdominowane.
                    if black3 is None:
                        black3 = black1|black2
                        grey3 = grey1|grey2
                        white3 = (white1 - grey2)|(white2 - grey1)
                    else:
                        b3 = black1|black2
                        g3 = grey1|grey2
                        w3 = (white1 - grey2)|(white2 - grey1)
                        if len(b3) < len(black3):   # mamy lepsze rozwiazanie
                            black3 = b3
                            grey3 = g3
                            white3 = w3
            result.append((white3, grey3, black3))
        return result

    def _visit(self, top):
        """Explore recursively the connected component."""
        # Start from a single node.
        # Tworze liste mozliwych rozwiazan dla top.
        # Trzeba sprawdzic wszystkie mozliwe kolorowania.
        # Zaczynamy od kolorowania dla samego worka top.
        # Wybieram rozne czarne, a szare same wychodza.
        # Ta czesc wyzyskuje informacje o krawedziach w worku.
        arg1 = []
        for subbag in iter_power_set(top):
            white = set()
            grey = set()
            black = set(subbag)
            # Szukamy szarych i bialych wierzcholkow.
            for source in top:
                if source in black:   # source is black
                    continue
                for target in self.graph.iteradjacent(source):
                    if target in black:   # source is grey
                        grey.add(source)
                        break
                if source not in grey:   # source is white
                    white.add(source)
            arg1.append((white, grey, black))   # kolorowanie
            # Kazdy wierzcholek gdzies musi nalezec.
            assert len(white) + len(grey) + len(black) == len(top)
        # Do zbiorow dominujacych dolaczamy wierzcholki od workow dzieci.
        for bag in self.td.iteradjacent(top):
            if bag not in self.parent:
                self.parent[bag] = top   # before _visit
```

```
                arg2 = self._visit(bag)
                arg1 = self._compose(top, arg1, bag, arg2)
        return arg1
```

5. Podsumowanie

Dekompozycja drzewowa jest przydatnym narzędziem obrazującym strukturę grafu i może być wykorzystana do szybkiego rozwiązywania trudnych problemów grafowych. Pewnym utrudnieniem jest znajdowanie optymalnej dekompozycji, bo to też jest problem trudny. Jednak w praktyce może wystarczyć znajomość dekompozycji niezbyt odległej od optymalnej, dlatego aktywnie szuka się coraz lepszych algorytmów przybliżonych. Z drugiej strony, ważna jest identyfikacja tych rodzin grafów, dla których można znaleźć optymalną dekompozycję w czasie wielomianowym (grafy cięciwowe, grafy Halina).

W niniejszej pracy zaimplementowano algorytm wyznaczający optymalną dekompozycję drzewową dla grafów cięciwowych. Na tej podstawie podano dwa sposoby wyznaczania (przybliżonej) dekompozycji dla ogólnych grafów. Pierwszy sposób polega na podaniu ciągu wierzchołków będącym PEO dla pewnego uzupełnienia cięciwowego danego grafu. Drugi sposób polega na znalezieniu uzupełnienia cięciwowego za pomocą heurystyki najmniejszego stopnia.

W ramach pracy przygotowano algorytmy wykorzystujące dekompozycję drzewową do wyznaczenia największego zbioru niezależnego, najmniejszego zbioru dominującego i najmniejszego pokrycia wierzchołkowego. Złożoność obliczeniowa algorytmów zależy liniowo od liczby wierzchołków, ale wykładniczo od szerokości drzewowej. Z tego powodu praktyczne zastosowanie tych algorytmów ogranicza się do grafów o małej szerokości drzewowej.

Dla grafów cięciwowych istnieje szybki algorytm o czasie liniowym do wyznaczenia największego zbioru niezależnego, gdzie korzysta się ze znanego PEO. Dla problemu zbioru dominującego i problemu pokrycia wierzchołkowego wykonano modyfikacje ogólnych algorytmów, wykonującą szybsze testy w każdym worku, bo każdy worek grafu cięciwowego jest kliką.

Wszystkie algorytmy zostały przetestowane pod względem poprawności i praktycznej złożoności obliczeniowej. Implementacje mogą być inspiracją do tworzenia nowych algorytmów wykorzystujących dekompozycję drzewową.

A. Testy algorytmów

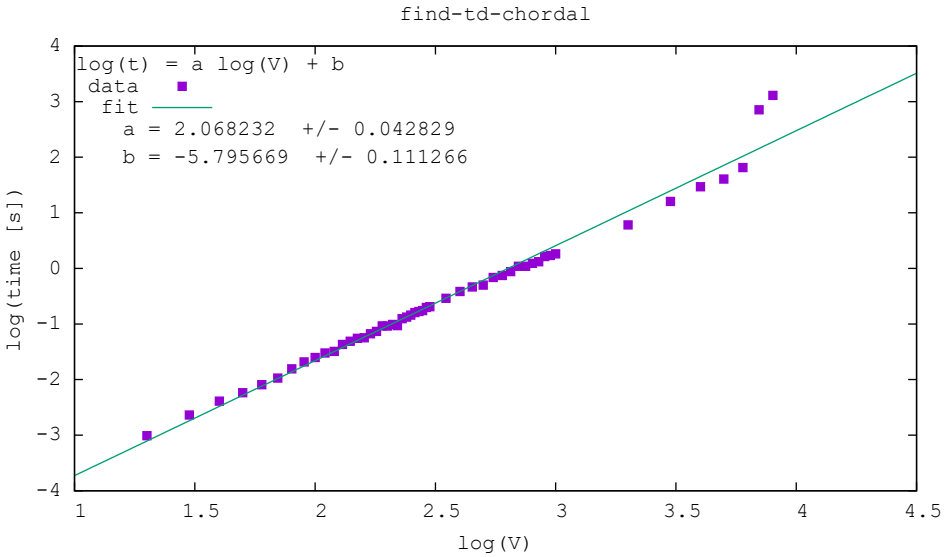
W niniejszym dodatku zebrano wyniki testów algorytmów, których implementacje powstały w tej pracy.

A.1. Testy dekompozycji drzewowej grafów cięciwowych

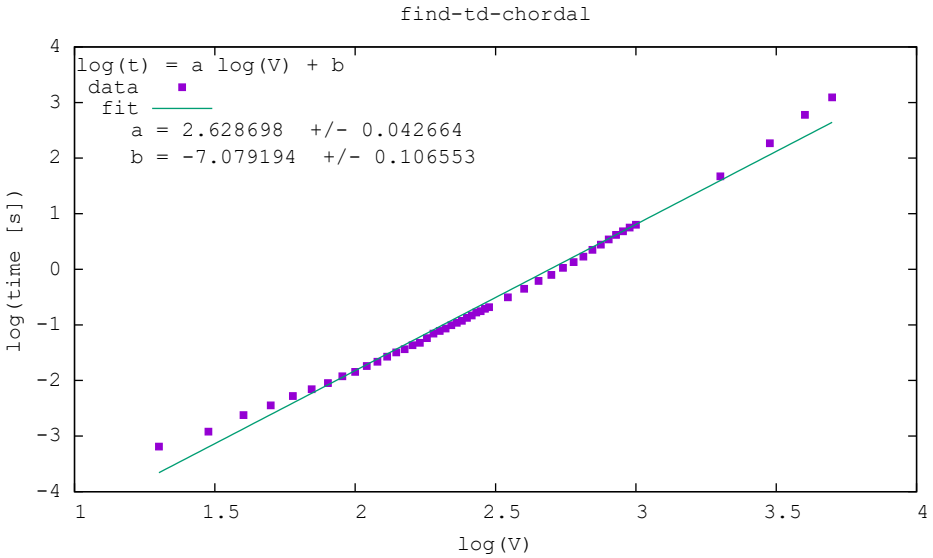
Testowanie wyznaczania dekompozycji drzewowej dla grafów cięciwowych, czyli funkcji find_td_chordal(). Sprawdzono eksperymentalnie czas pracy algorytmu dla k-drzew z n wierzchołkami, gdzie $k = 5$ (wykres A.1) lub $k = n/2$ (wykres A.2). Testy potwierdzają złożoność $O(n^2k)$. Dla małych k zaobserwowaliśmy słaby wzrost współczynnika regresji a ze wzrostem k.

A.2. Testy zbiorów niezależnych

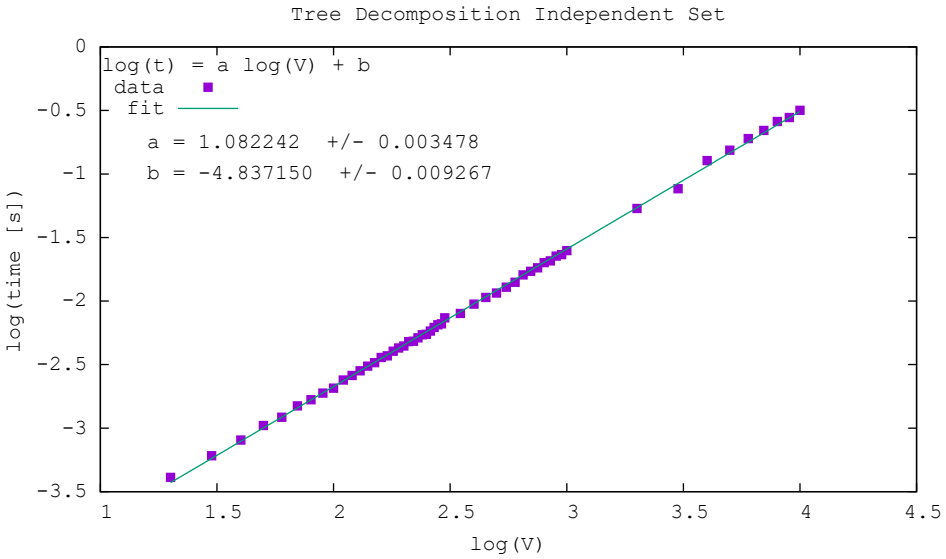
Testowanie wyznaczania największego zbioru niezależnego dla grafów o danej dekompozycji drzewowej, czyli klasy TDIndependentSet. Sprawdzono eksperymentalnie czas pracy algorytmu dla przypadkowych k-drzew, $k = 2, \ldots, 10$. Wykresy potwierdzają liniową zależność czasu pracy od liczby wierzchołków dla ustalonego k. Dla rosnącego k obserwuje się wzrost czasu pracy w przybliżeniu o stały czynnik, co potwierdza zależność wykładniczą od szerokości drzewowej k (wykres A.12).



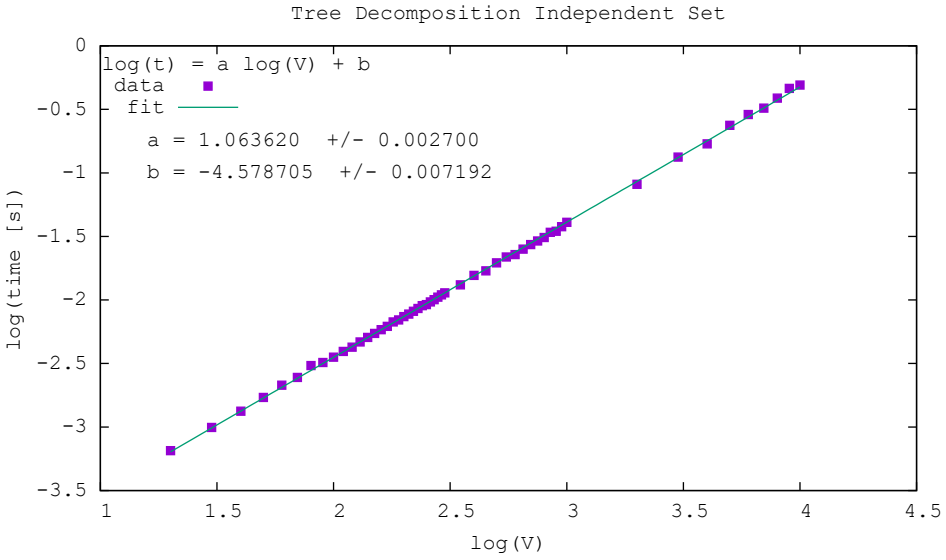
Rysunek A.1. Wykres przedstawiający wydajność algorytmu wyznaczania dekompozycji drzewowej dla k-drzewa z $k = 5$. Współczynnik a bliski 2 potwierdza zależność $O(n^2k)$, ponieważ k jest stałe.



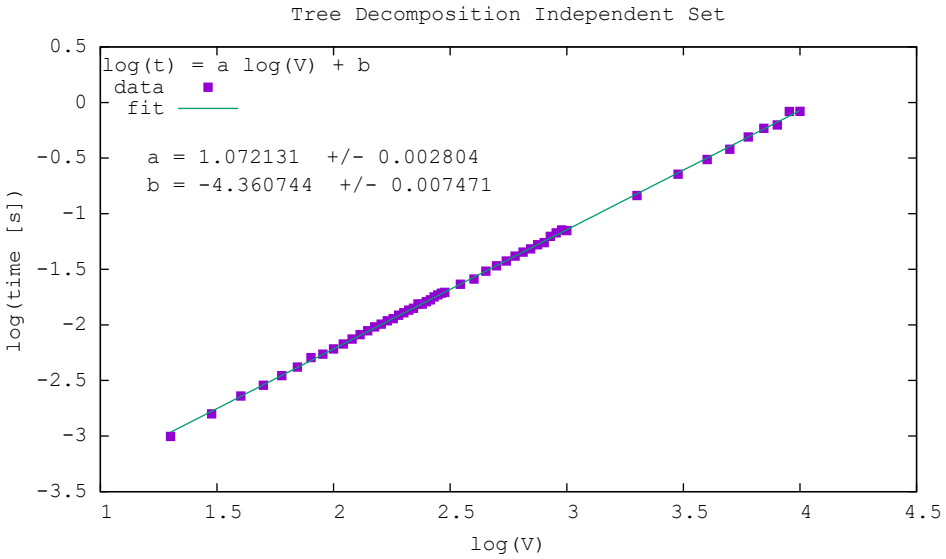
Rysunek A.2. Wykres przedstawiający wydajność algorytmu wyznaczania dekompozycji drzewowej dla k-drzewa z $k = n/2$. Współczynnik a leżący między 2.5 a 3 potwierdza zależność $O(n^2k)$, ponieważ k rośnie liniowo z n.



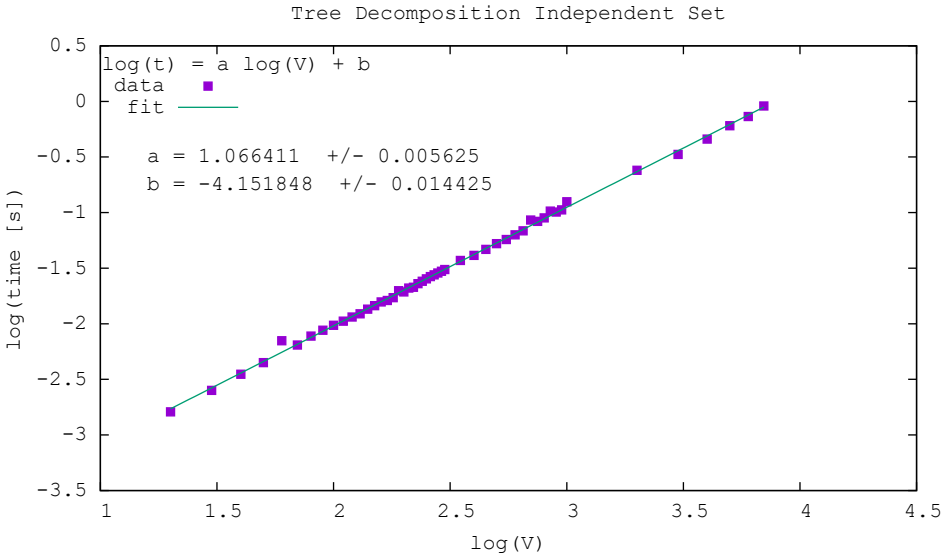
Rysunek A.3. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 2$.



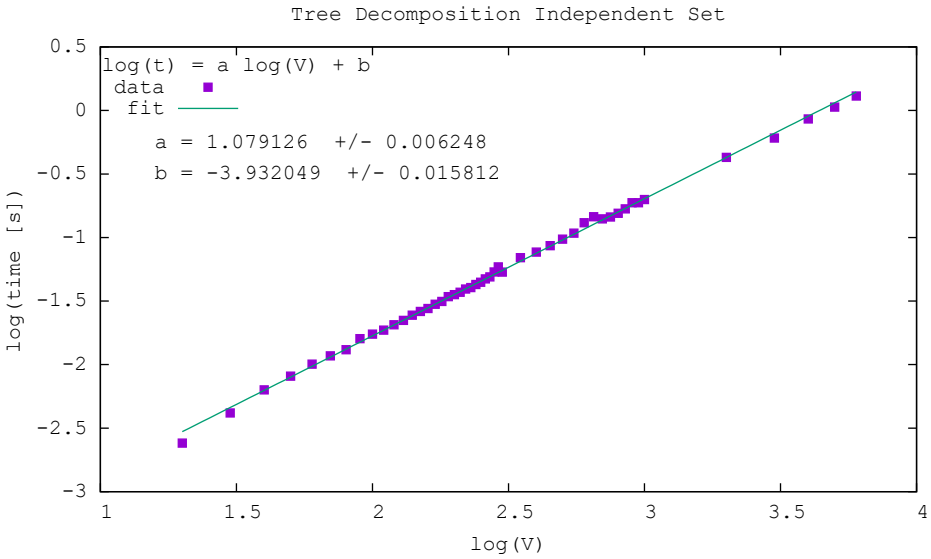
Rysunek A.4. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 3$.



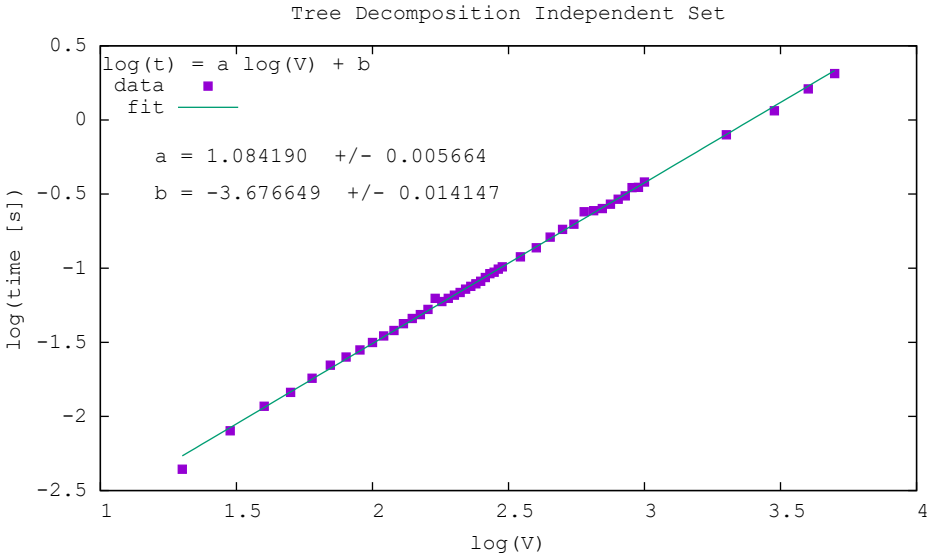
Rysunek A.5. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 4$.



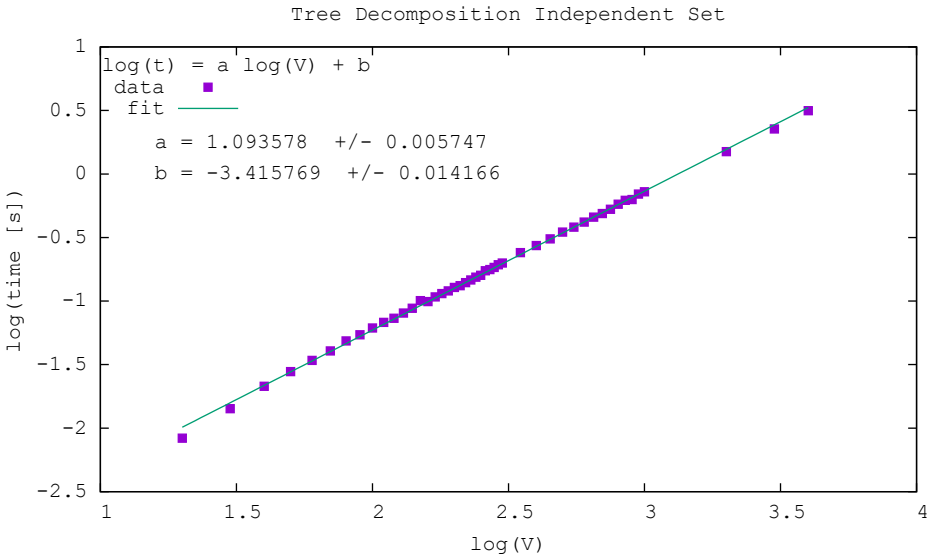
Rysunek A.6. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dlak $k = 5$.



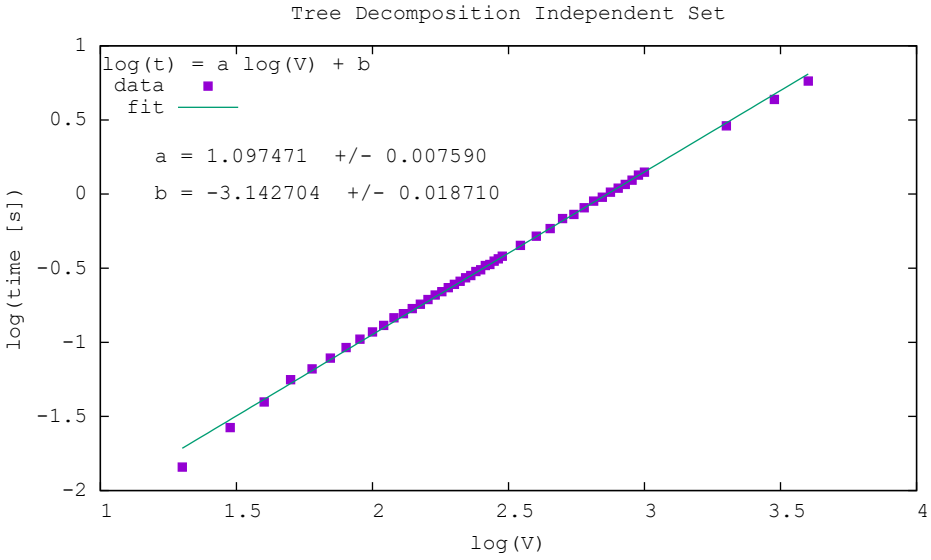
Rysunek A.7. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 6$.



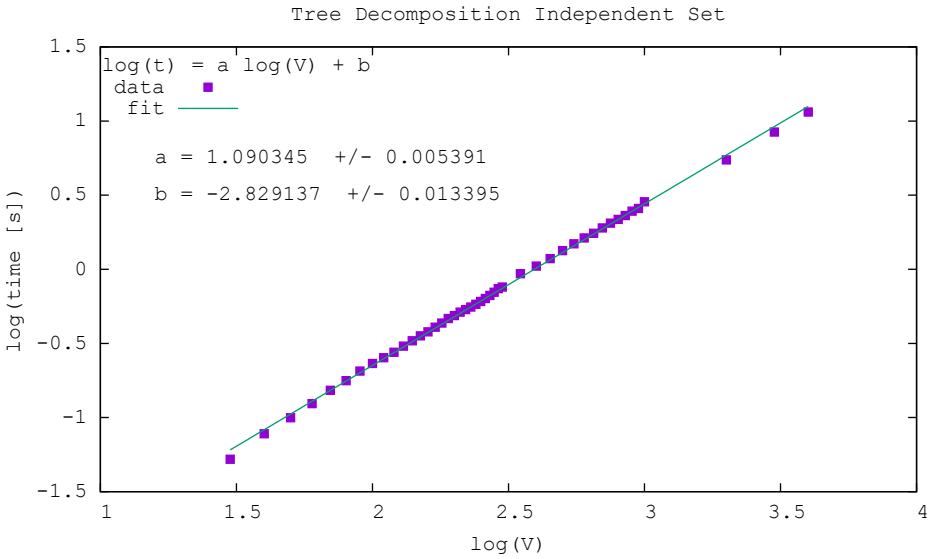
Rysunek A.8. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 7$.



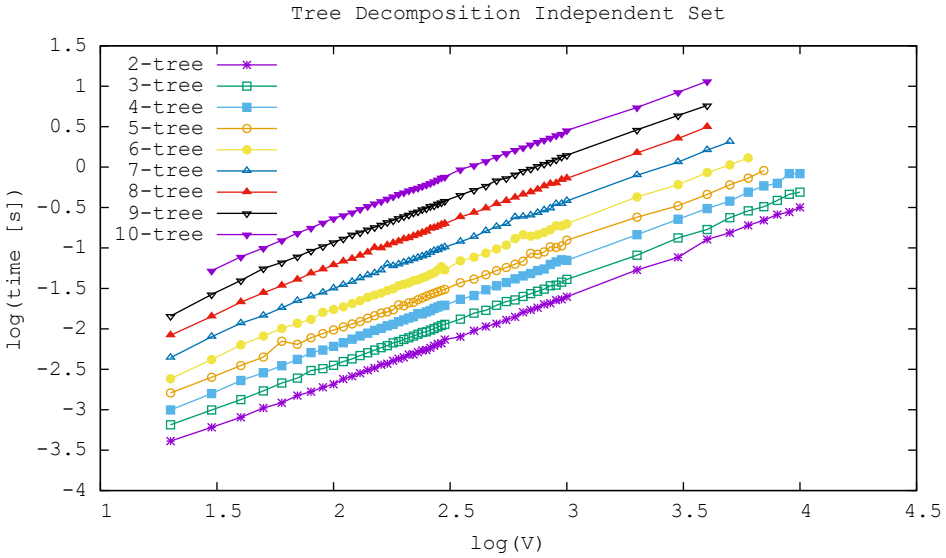
Rysunek A.9. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 8$.



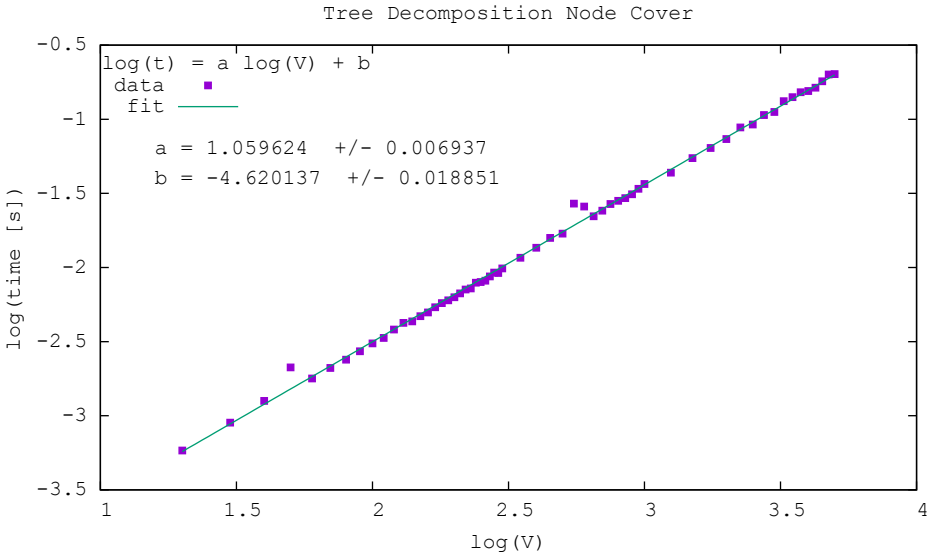
Rysunek A.10. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 9$.



Rysunek A.11. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego dla $k = 10$.



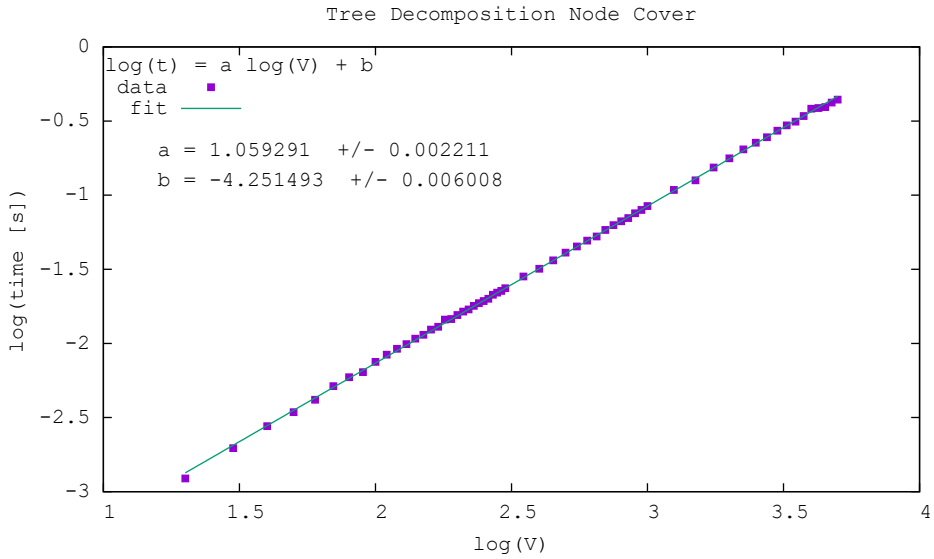
Rysunek A.12. Wykres przedstawiający wydajność algorytmu wyznaczania największego zbioru niezależnego w zależności od parametru k.



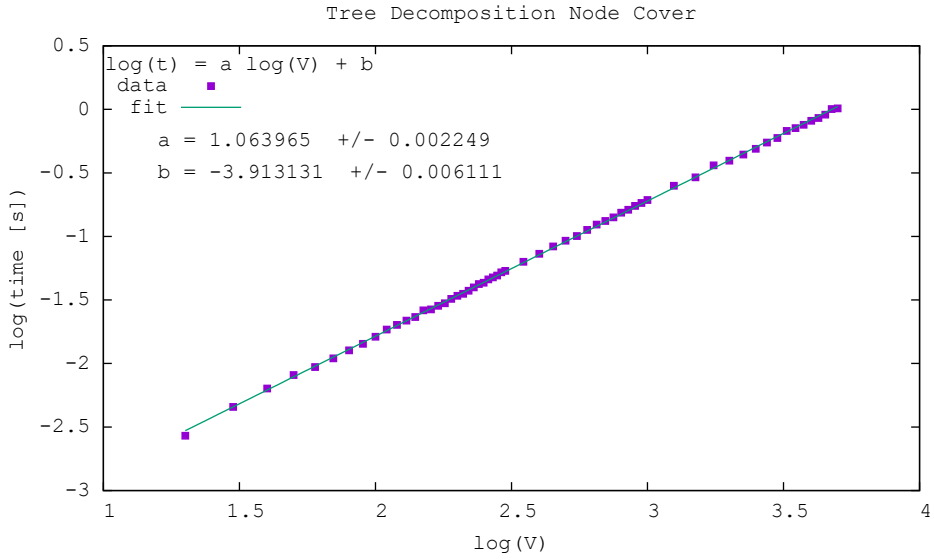
Rysunek A.13. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 2$, widoczna zależność liniowa.

A.3. Testy pokrycia wierzchołkowego

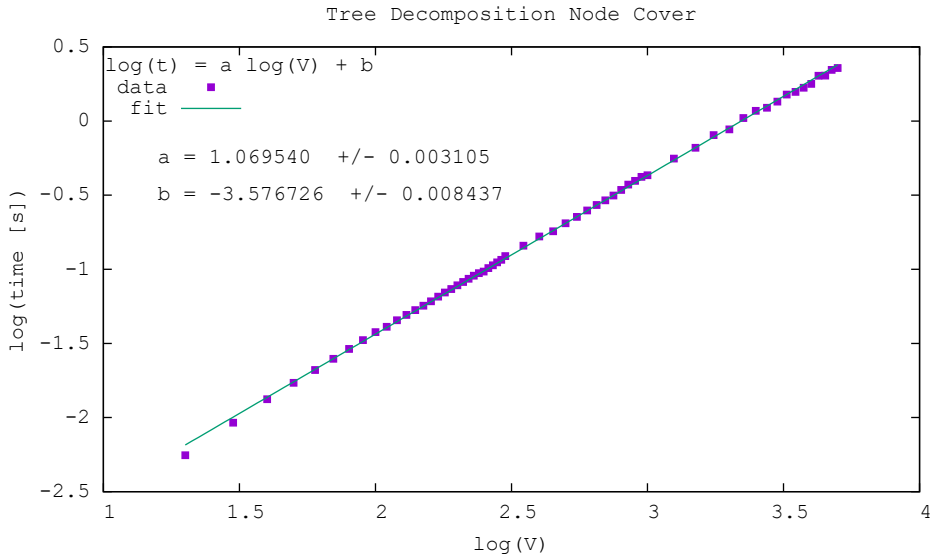
Testowanie wyznaczania najmniejszego pokrycia wierzchołkowego dla grafów o danej dekompozycji drzewowej, czyli klasy TDNodeCover. Sprawdzono eksperymentalnie czas pracy algorytmu dla przypadkowych k-drzew, $k = 2, \ldots, 9$. Wykresy potwierdzają liniową zależność czasu pracy od liczby wierzchołków dla ustalonego k. Dla rosnącego k obserwuje się wzrost czasu pracy w przybliżeniu o stały czynnik, co potwierdza zależność wykładniczą od szerokości drzewowej k (wykres A.21).



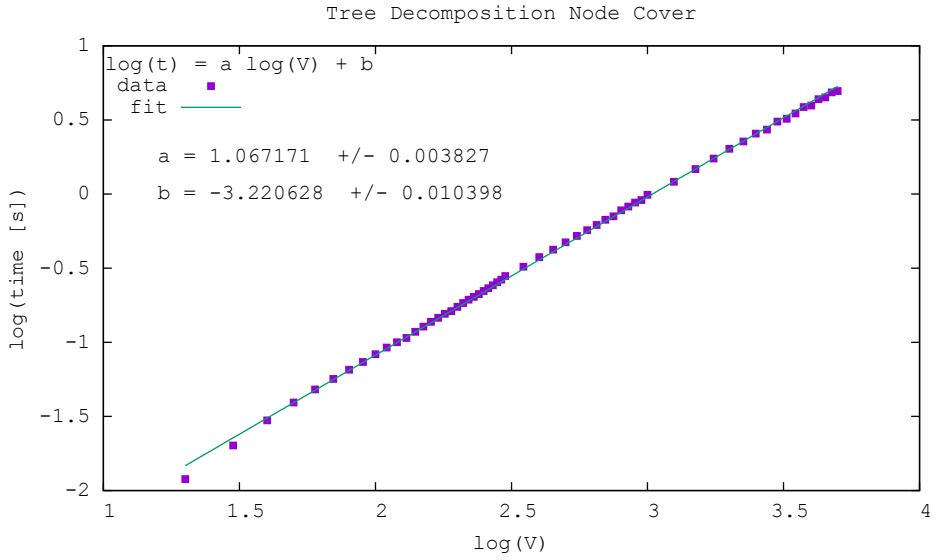
Rysunek A.14. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 3$, widoczna zależność liniowa.



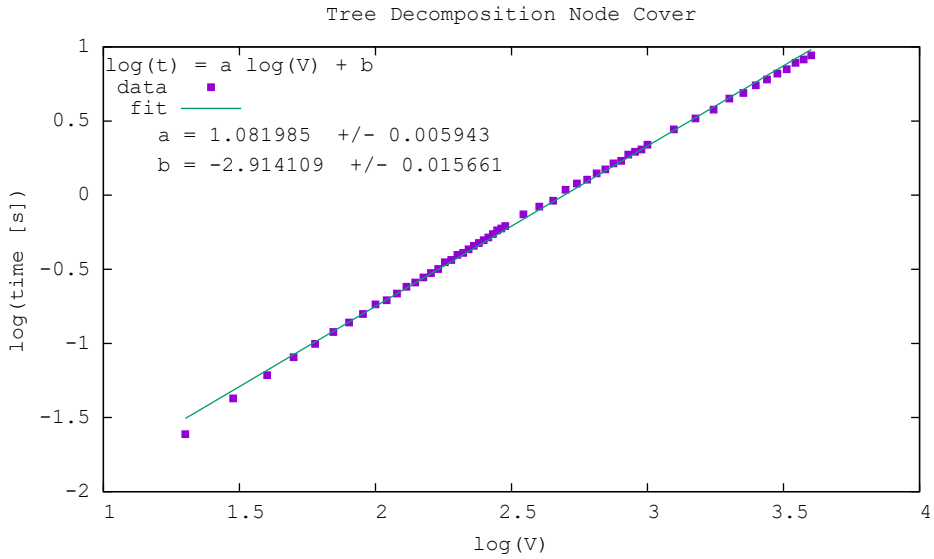
Rysunek A.15. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 4$, widoczna zależność liniowa.



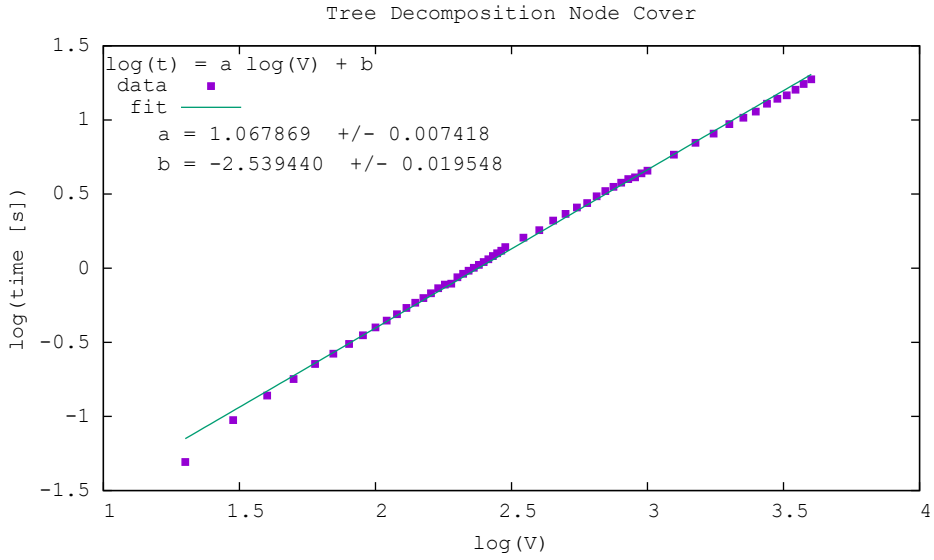
Rysunek A.16. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 5$, widoczna zależność liniowa.



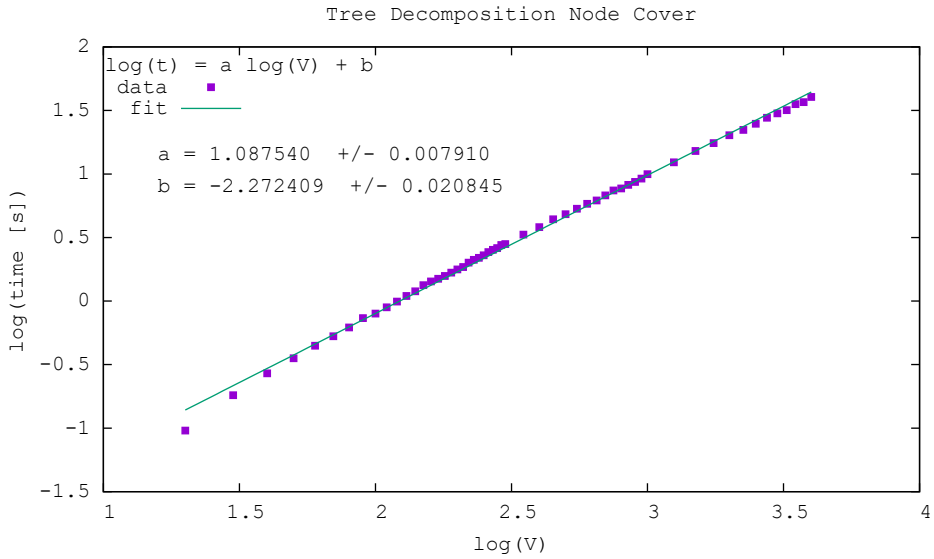
Rysunek A.17. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 6$, widoczna zależność liniowa.



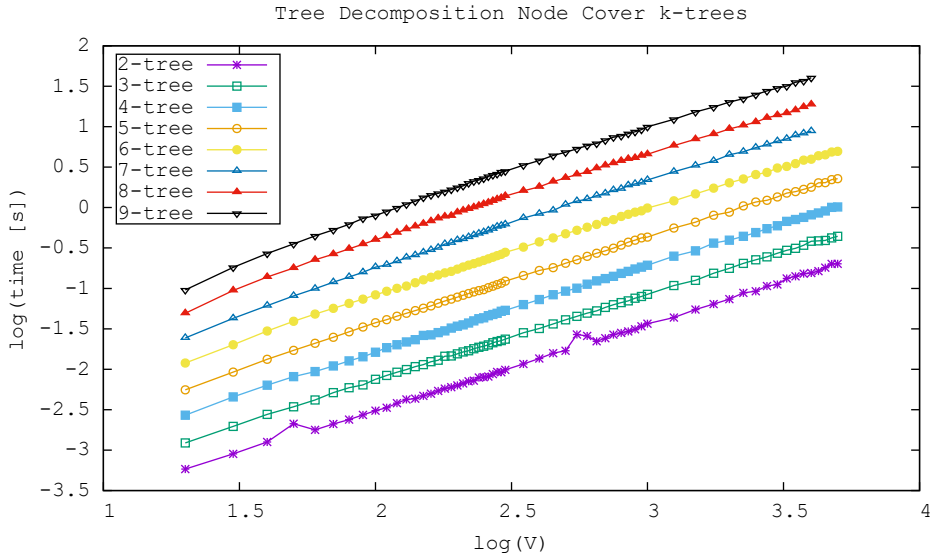
Rysunek A.18. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 7$, widoczna zależność liniowa.



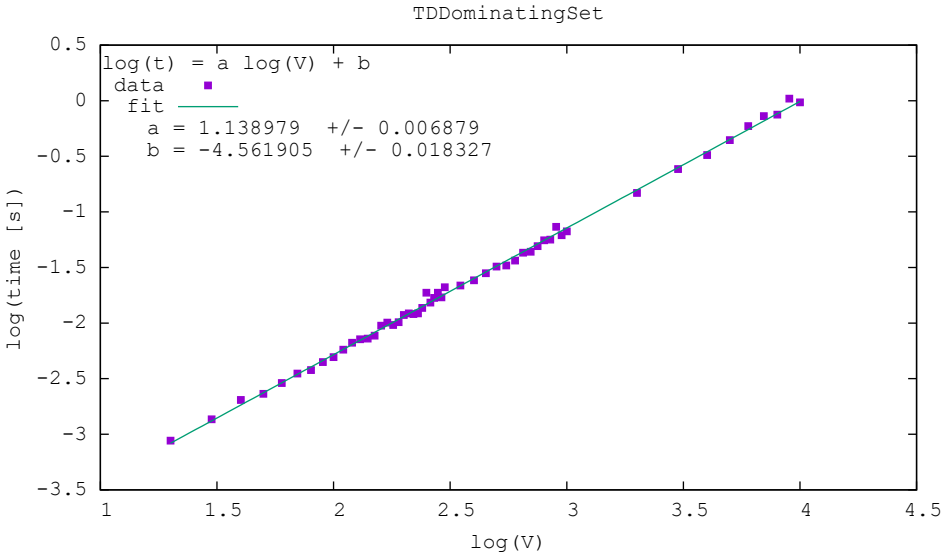
Rysunek A.19. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 8$, widoczna zależność liniowa.



Rysunek A.20. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego pokrycia wierzchołkowego dla $k = 9$, widoczna zależność liniowa.



Rysunek A.21. Wykres przedstawiający wydajność algorytmu TDNodeCover w zależności od parametru k. Czas pracy algorytmu rośnie liniowo z k.

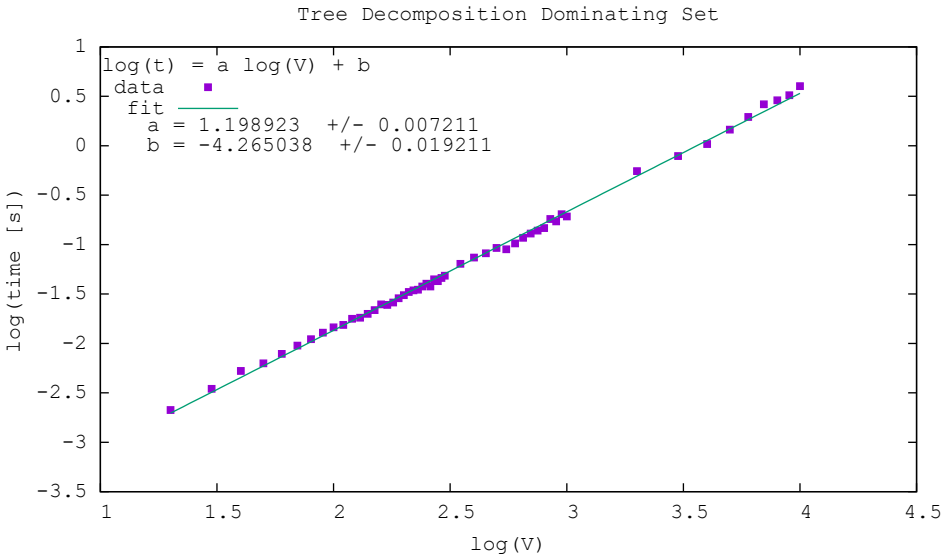


Rysunek A.22. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 2$.

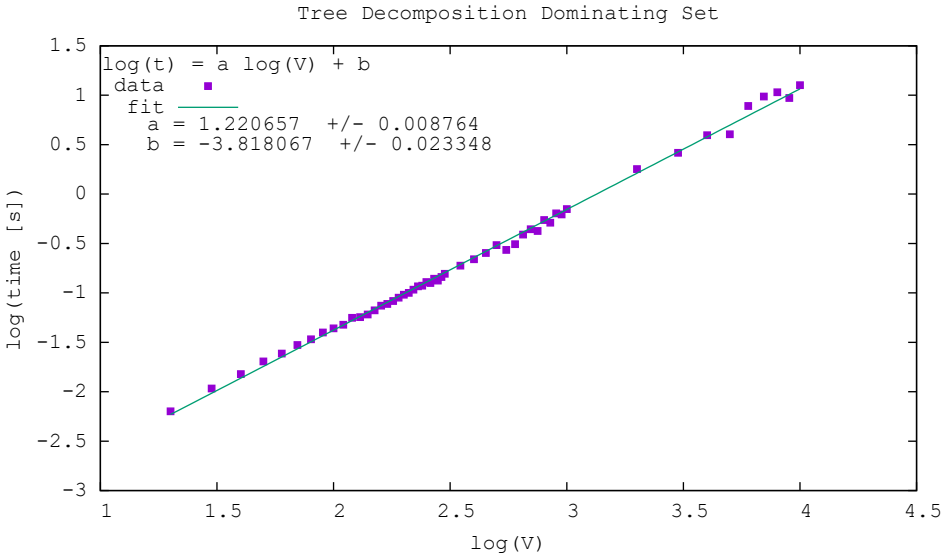
A.4. Testy zbiorów dominujących

Testowanie wyznaczania najmniejszego zbioru dominującego dla grafów o danej dekompozycji drzewowej, czyli klasy TDDominatingSet. Sprawdzono eksperymentalnie czas pracy algorytmu dla przypadkowych k-drzew, $k = 2, \ldots, 9$. Wykresy potwierdzają liniową zależność czasu pracy od liczby wierzchołków dla ustalonego k. Dla rosnącego k obserwuje się wzrost czasu pracy w przybliżeniu o stały czynnik, co potwierdza zależność wykładniczą od szerokości drzewowej k (wykres A.30).

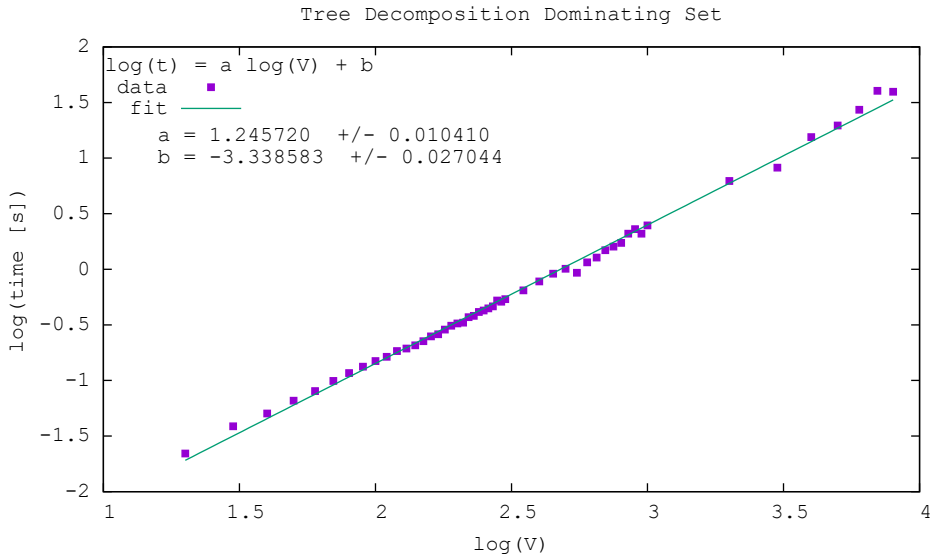
Zakładając że spodziewany czas pracy algorytmów ma czynnik wykładniczy z potegą k, a dokładniej 4^k (iset, cover) lub 9^k (dset), zostały sporządzone wykresy zależnosci skoku czasu o stały wspołczynnik b_k dla kolejnych parametrów k. Otrzymane doświadczalnie wyniki współczynnika u podstawy to odpowiednio 1.75(2) [iset], 2.17(2) [cover], 3.66(23) [dset]. Współczynniki wydają się być dość małe, co może wynikać z faktu, że testy przeprowadzono na k-drzewach. Możliwe, że inne grafy wymagają dłuższych obliczeń.



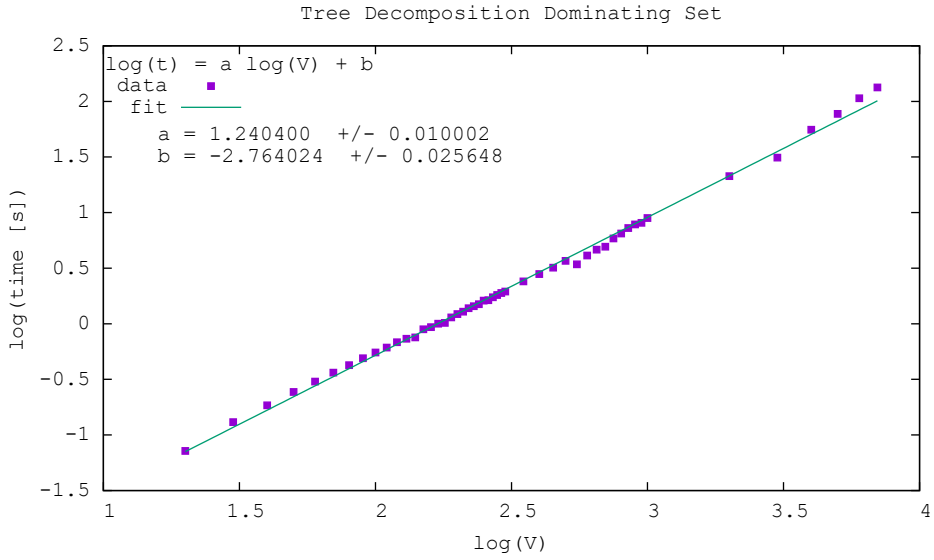
Rysunek A.23. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 3$.



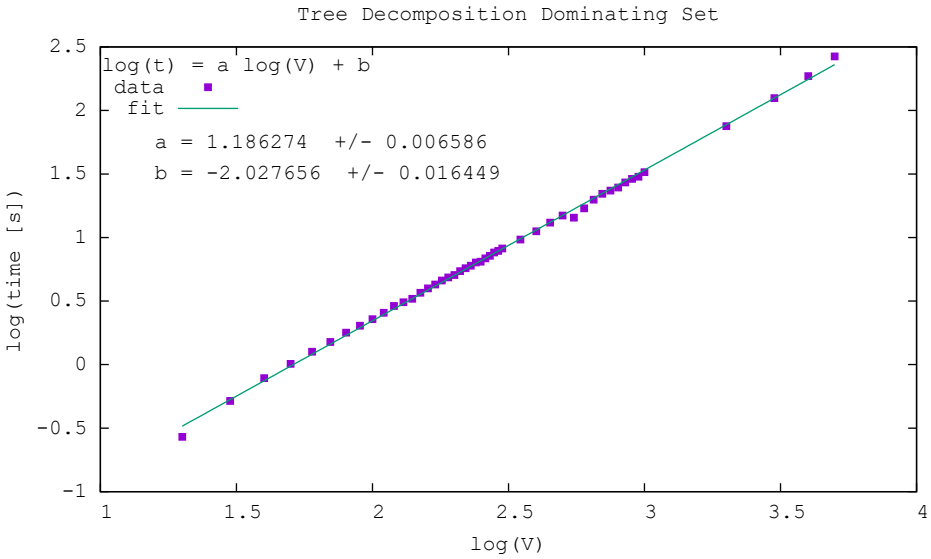
Rysunek A.24. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 4$.



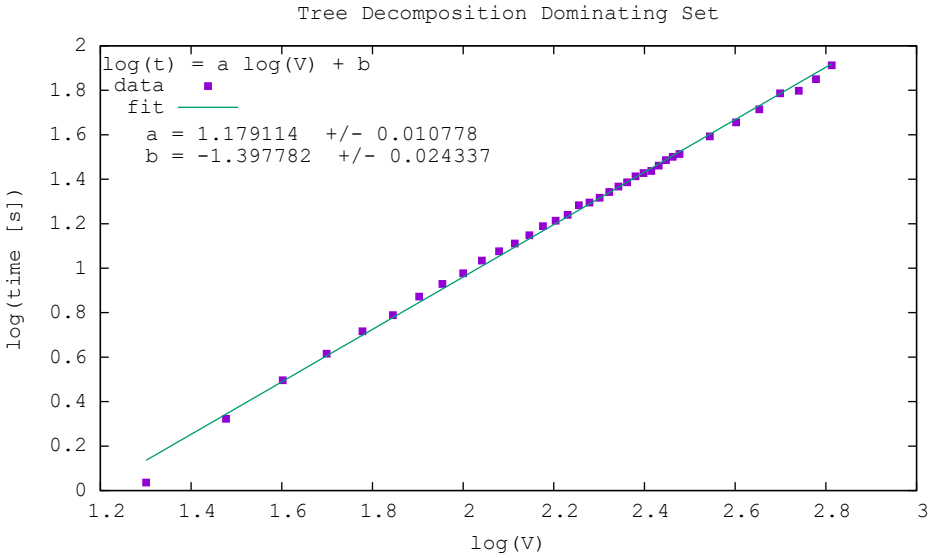
Rysunek A.25. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 5$.



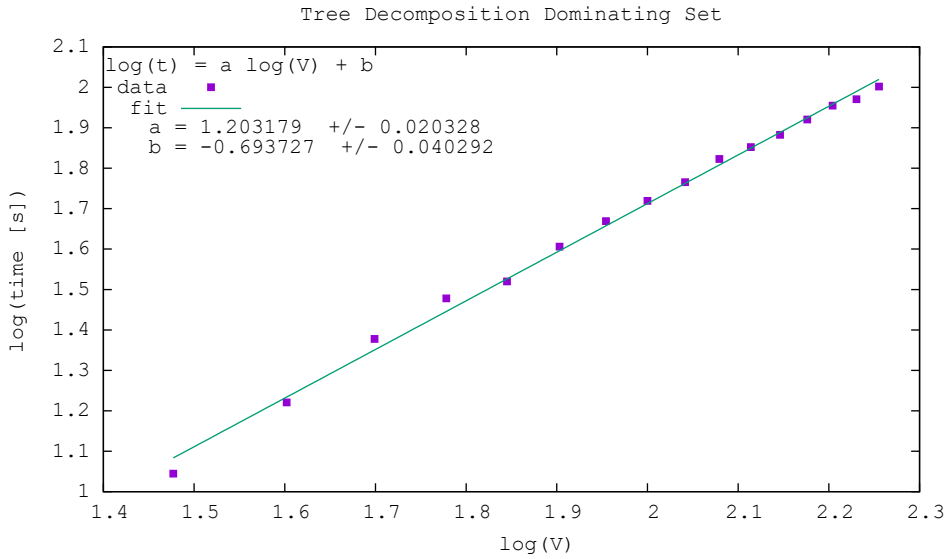
Rysunek A.26. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 6$.



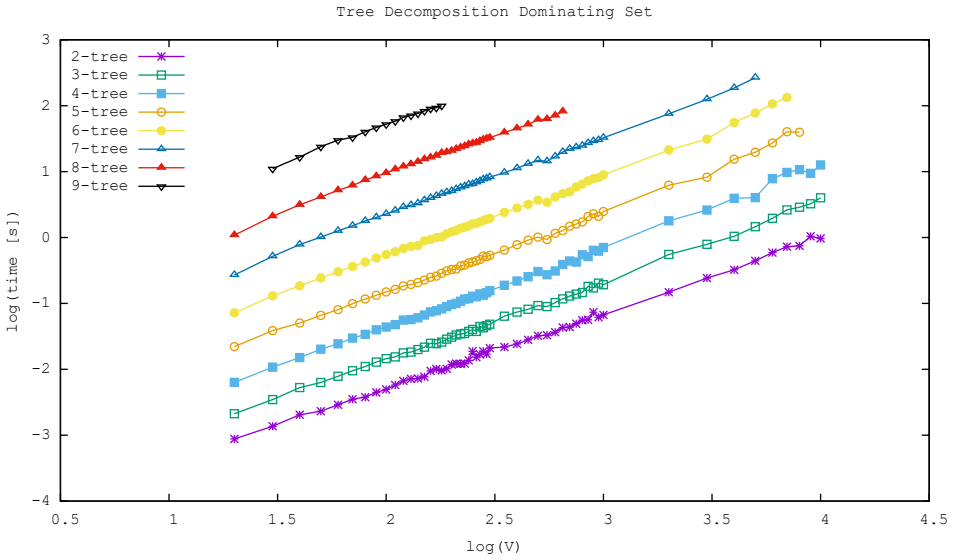
Rysunek A.27. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 7$.



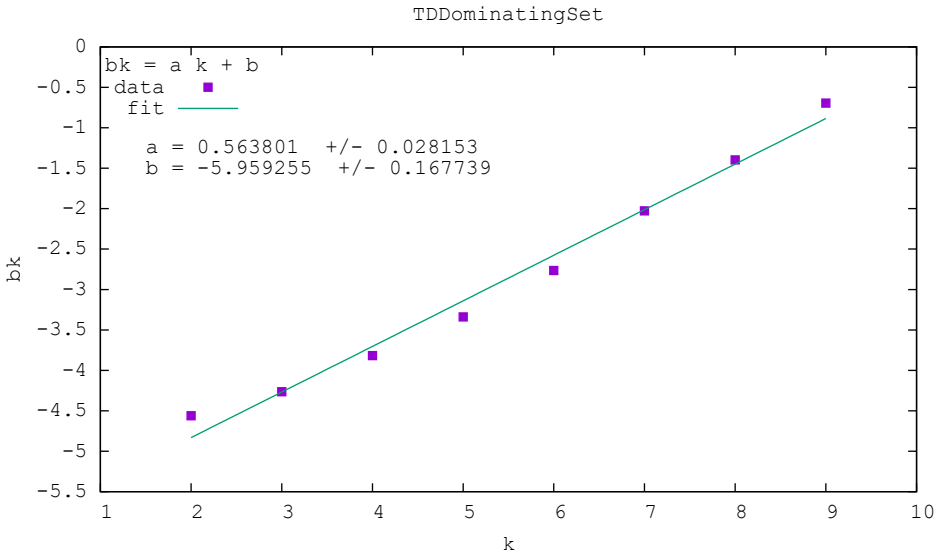
Rysunek A.28. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 8$.



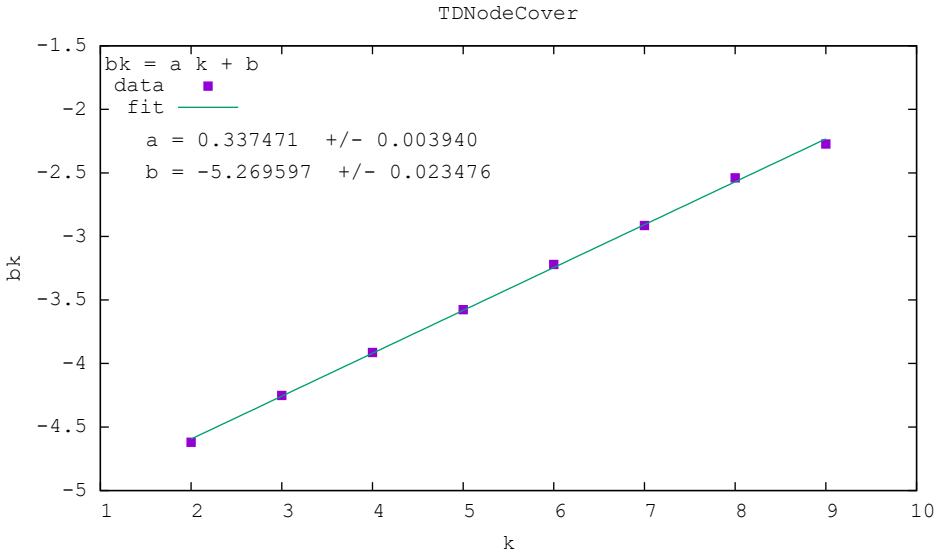
Rysunek A.29. Wykres przedstawiający wydajność algorytmu wyznaczania najmniejszego zbioru dominującego dla $k = 9$.



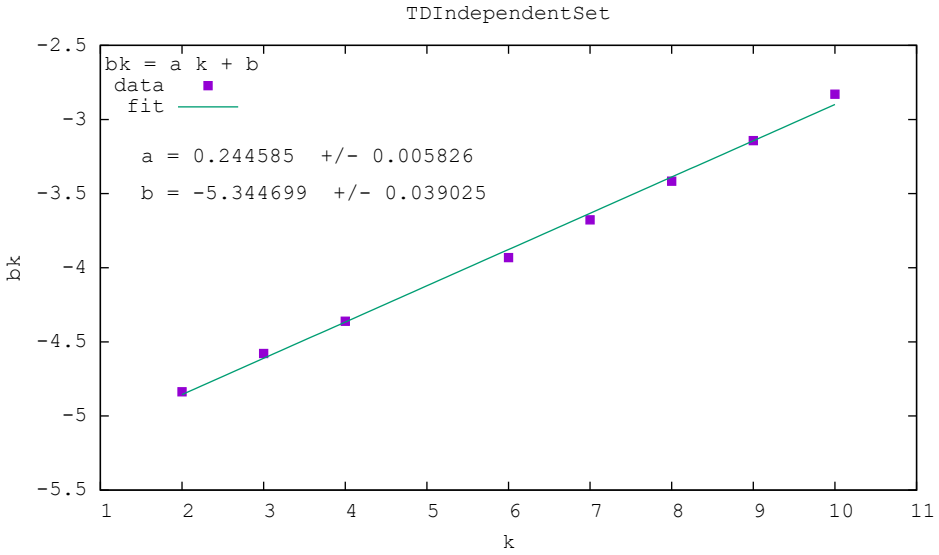
Rysunek A.30. Wykres porównania wydajności algorytmu TDDominatingSet w zależności od parametru k. Czas pracy algorytmu rośnie liniowo z k.



Rysunek A.31. Wykres zależności zmiany czasu pracy algorytmu TDDominating-Set o stały czynnik b_k w zależności od parametru.



Rysunek A.32. Wykres zależności zmiany czasu pracy algorytmu TDNodeCover o stały czynnik b_k w zależności od parametru k.



Rysunek A.33. Wykres zależności zmiany czasu pracy algorytmu TDIndependentSet o stały czynnik b_k w zależności od parametru k.

Bibliografia

[1] Wikipedia, Tree decomposition, 2019,
`https://en.wikipedia.org/wiki/Tree_decomposition`.

[2] Wikipedia, Treewidth, 2019,
`https://en.wikipedia.org/wiki/Treewidth`.

[3] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, Saket Saurabh, *Parameterized Algorithms*, Springer 2015.
`http://parameterized-algorithms.mimuw.edu.pl/`.

[4] Rim van Wersch, Steven Kelk, *ToTo: An open database for computation, storage and retrieval of tree decompositions*, Discrete Applied Mathematics 217, 389-393 (2017).
`http://treedecompositions.com/`.

[5] Gnuplot, Documentation
`https://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html`.

[6] Wikipedia, K-tree, 2019,
`https://en.wikipedia.org/wiki/K-tree`

[7] Python Programming Language - Official Website,
`https://www.python.org/`.

[8] Andrzej Kapanowski, graphs-dict, GitHub repository, 2019,
`https://github.com/ufkapano/graphs-dict/`.

[9] Serwis *TeXample.net*, Documentacja dla pakietu TikZ, 2019,
`http://www.texample.net/tikz/`.

[10] Martin Charles Golumbic, *Algorithmic Graph Theory and Perfect Graphs, Second Edition*, Annals of Discrete Mathematics, Volume 57, Elsevier 2004 [First edition 1980].

[11] Małgorzata Olak, *Badanie grafów cięciwowych w języku Python*, Praca magisterska, Uniwersytet Jagielloński, Kraków, 2017.

[12] Jean R. S. Blair, Barry Peyton, *An introduction to chordal graphs and clique trees*, In: A. George, J. R. Gilbert, J. W. H. Liu (eds), Graph Theory and Sparse Matrix Computation, The IMA Volumes in Mathematics and its Applications, Springer-Verlag, New York, NY, Volume 56, pp. 1-29, 1993.

[13] Fanica Gavril, *Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph*, SIAM Journal on Computing 1(2), 180-187 (1972).

[14] Aleksander Krawczyk, *Badanie grafów Halina z językiem Python*, Uniwersytet Jagielloński, Kraków, 2016.

[15] Konrad Gałuszka, *Badanie grafów szeregowo-równoległych z językiem Python*, Uniwersytet Jagielloński, Kraków, 2018.